RHYTHM®

CUSTOMIZATION MANUAL

supply chain planner

i2 Technologies

Concepts / Reference

RHYTHM SCP

Customization Manual

Contents

RHYTHM® SCP
Customization Manual

Printed in the United States of America.

i2 TECHNOLOGIES, INC.
909 East Las Colinas Blvd.
16th Floor
Irving, Texas 75039
USA

March 27, 1998

Table of Contents

List of Figures

Section 1 Introduction

The process of implementing RHYTHM Supply Chain Planner (SCP) to deliver the required behavior and business benefits is *customization*. To do this, an implementor is engaged in multiple tasks including:

- organizing the file directory structure that will hold the various elements required by RHYTHM SCP servers and clients
- providing system files to define implementation-specific model fields, client profiles (reports), and so on
- providing reports files that specify the data and views available to each client
- specifying and implementing the mapping of data from external sources into RHYTHM SCP models and vice versa
- developing the necessary client interfaces (interactive or non-interactive) to perform planning functions on the RHYTHM SCP server
- adjusting interactive client interfaces to support end-user workflows

The goal of the RHYTHM SCP Object Interaction Language User Manual is to provide "one-stop shopping" for implementors to find and learn what they need to perform these tasks. This manual consists of user and reference materials designed to supplement, but not replace, the RHYTHM SCP Model Reference Manual.

1.1 Object Interaction Language

Object Interaction Language (OIL) is the primary tool for implementors of RHYTHM SCP, and as such, is the primary topic of this manual.

OIL is an interpreted, functional language. Its functions and expressions are compiled and checked when read by RHYTHM SCP, and all expressions can be used interactively. OIL allows access to and manipulation of the model information residing on the server through its worksheets. OIL worksheets are designed to use the "spreadsheet" paradigm as established by other software packages. OIL maps to, and utilizes, RHYTHM SCP model definitions and their interrelationships. It also allows parameterization of interfaces: interfaces can be tied to types of models (e.g. SITE) rather than to specific instances of models (e.g. Joe's Site).

OIL is a strongly *typed* programming language. All function invocations expect arguments of a definite *type*, and all results from expressions and function calls are of definite *types*. OIL provides utilities for conversion from one *type* to another. These utilities (built into the language) are both automatic and explicit. OIL optimizes the execution of routines based on the context of execution and the availability of the data.

1.2 Overview of OIL Elements

OIL provides various elements for computing data for display to you, and for presenting the data in different ways for you to view. Several of these elements are listed below:

- Reports
- Worksheets
- Layouts
- Functional Worksheets
- do_files

1.2.1 Reports, Layouts, Worksheets: As OIL Elements

A report is a mechanism that provides information about the RHYTHM SCP models running on the RHYTHM SCP engine. Each report is a container for one or more layouts. Each layout in a report describes the display of data contained in a worksheet.

A report can be thought of as a two-dimensional table of information. The *Layouts* describe rows and columns of the two-dimensional table. Layouts are arranged vertically or horizontally. Each *layout* is associated with exactly one *worksheet* which contains calculations for the data in the *layout*. Each element of a layout is called a control. The control has a corresponding cell in a worksheet. Different *reports* can use the same *layout*, and *worksheets* can hold data for different *layouts*.

Reports, layouts, and worksheets are stored as ASCII text files. These files are processed by the RHYTHM SCP engine to create the actual mechanisms described by these text files. Changes to reports, layouts, and 'worksheets are achieved by changing the contents of these text files, and having the RHYTHM SCP engine process the files that have changed.

1.2.2 Data Computation

The following concepts describe the elements involved in computing data for reports:

- *Worksheet* - A worksheet consists of a table of cells serving as the computational element of the report mechanism. For the purpose of identifying the cells, columns are specified using a letter of the alphabet and rows are specified using a number beginning with 1. A single worksheet may be shared by multiple layouts. Worksheets are either *normal*, *replicating*, or *functional* as explained below.
- *Cell* - A cell contains an expression to display a result (the *get* expression) and an expression to store a value (the *set* expression). The expression may compute a single value (can be stored in a normal cell) or multiple values (can be stored in a replicating cell).
    - normal - get expression and set expression must return a single atomic value (not a list)
    - functional (procedure) - get expression can return both atomic values and lists
    - replicating - get and set expressions always return lists, each element of which resides on a "layer"

  Every cell is referenced by a control in the layout that refers to the worksheet.
- *Expression* - An expression is a formula consisting of operators, functions, constants, variables, and parameters. Expressions are *typed* in the sense that there is a data type associated with each parameter, variable, constant, and function return value. Expressions may refer to internal models and their fields.

1.3 Directory Structure

RHYTHM SCP searches directory paths to find worksheets, layouts, and reports, as well as system setup files, data import files, saved binary images, and so on. There are several *environment variables* that determine the directory path RHYTHM SCP searches. These variables are set by corresponding RHYTHM SCP engine invocation options, such as:

- *data* - specifies the pathname to the directory that contains data import files for the RHYTHM SCP engine.
- *reports* - specifies the pathname to the subdirectories that contain all the available worksheets, reports, and layouts.
- *system* - specifies the pathname to the directory that contains system level data import files for the RHYTHM SCP engine.
- *include* - specifies other pathnames to be included in the search path.

Every client that connects to the RHYTHM SCP engine will have associated with it a particular directory path sequence to be searched to find the subset of worksheets, layouts, and reports associated with that client (the files in the system directory in part define this directory path sequence for every known client or user). RHYTHM SCP searches the directory path in sequence and uses the first occurrence of the file found. For example, if you have a pathname structure of *alpha/beta/gamma*, and a report references a layout that exists in *gamma* named *layout_file*, then the file *gamma/layout_file* may be overridden by a file in *alpha* or in *beta* that is named *layout_file*.

1.4 Data Types

RHYTHM SCP uses data types to give meaning to its data (e.g. is 123 the integer one-hundred twenty-three or is it the string containing the numerals 1, 2, 3?). Types common to most programming languages are also available in OIL (e.g. logical, number). Application-specific types are also built-in (such as *date_range*) which are important for planning. A complete listing of all data types is provided in the RHYTHM SCP Model Reference Manual.

Every type has an associated format which specifies how data elements of the type are to appear (e.g. how many decimal places of a floating point number are to be displayed). Multiple formats may be supported for the particular type based on the context and use of that type. For example, *date* might have the day first on import, but the month first on export.

Section 2

Setup and Customization

This section provides you with basic background information on OIL. By reading the material, you will have the basis for creating reports using various components of OIL. You will also have the basic knowledge for obtaining RHYTHM SCP planning data using OIL.

2.1 OIL Overview

OIL is the framework for the client interface and all customizations of RHYTHM SCP. As such, OIL is the primary implementor's tool. With OIL, a single framework is available for the following:

- to map data from external sources to RHYTHM SCP models
- to develop the necessary client interfaces (interactive or non-interactive) to perform the planning function on the server

OIL is a *strongly typed* programming language. This implies the following attributes:

- All function invocations expect arguments of a definite *type*.
- Results from expressions and function calls are of definite *types*.

Utilities for conversion from one *type* to another are built into the language (both automatic and explicit). OIL optimizes the execution of routines based on the context of the execution and the availability of data.

OIL allows for the parameterization of interfaces. Interfaces can be tied to types of models (e.g. *Site*) instead of specific instances of models (e.g. *Joe's Site*).

OIL also has limited built-in support for GUI utilities (widgets, icons, etc.).

2.1.1 Key Definitions

Table 1 lists several key definitions that are important when learning OIL.

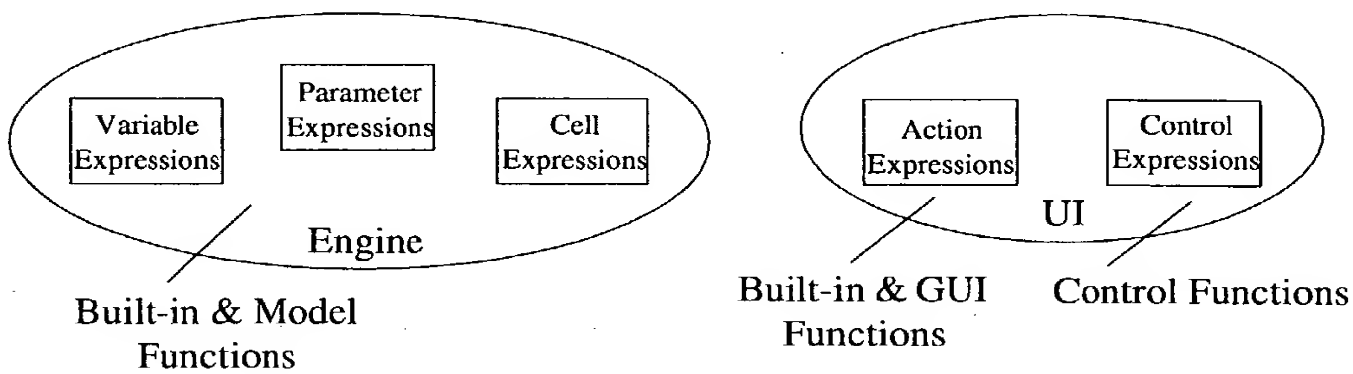
Table 1: Key Definitions

Term	Definition
type	*Type* is RHYTHM SCP's method of giving meaning to its data (e.g. is 123 the integer one-hundred twenty-three or is it the string containing the numerals 1, 2, and 3?). *Types* common to most programming languages are also available in OIL (e.g. logical, number). Each RHYTHM SCP model also constitutes a *type*. Application specific *types* are also built-in, such as *date_range*, which are important for planning. Refer to the *RHYTHM SCP Model Reference Manual* for a description of all named *types*.
formats	Every named type has an associated *format* which specifies how data elements of the type are to appear (e.g. how many decimal places of a floating point number are to be displayed). Multiple *formats* can be supported for a particular *type* based on the context and use of the *type* (e.g. *date* might have the day first on import, but the month first on export).
field	Each RHYTHM SCP model has one or more *fields* that together implement the model's behavior. For example, the *name field* in the *supply_chain* model. *Fields* have types as well. For instance, *name* is a *symbol*.
list	A *list* is a collection of RHYTHM SCP data of the same type. Each *list* element has a position in the *list*, but *list* elements are not ordered unless explicitly sorted. Special OIL functions are available for operating on *lists*.
expression	An *expression* is a formula, in OIL syntax, consisting of one or more the following elements: • constants • parameters • variables and computes • functions • operators • model fields *Expressions* return a typed result. The software parses the formula to ensure its syntactic accuracy, compiles it into machine-executable code, then executes that code.

2.1.1.1 Expressions: Engine vs. UI

Engine and UI expressions are distinguished by where the expression is evaluated. FIGURE 1 illustrates this distinction:

FIGURE 1 Engine and UI Expressions



Any function that needs to access fields or information on the engine must be evaluated on the engine.

2.1.2 Reports - What are they?

Reports are views of the planning data in context, given a user, a system environment, and input parameters. They include hardcopy printouts as well as GUI screens. *Reports* are created and managed with OIL.

Reports are also models. Each User (also a model) can have its own set of custom *reports* (written for that implementation) and standard *reports* (written by i2). RHYTHM SCP defines environment variables that help determine how a given RHYTHM SCP client gets reports for its use.

2.2 Setup and Customization Overview

The RHYTHM SCP server (*scp_engine*) is invoked from either a UNIX or Windows NT directory structure. The OIL client (*scp_ui* - batch or interactive, or *scp_batch*) is invoked from a Windows NT directory structure. The system setup information for the application is also located in that same structure.

2.2.1 System Setup

The directory structure distributed with the system includes:

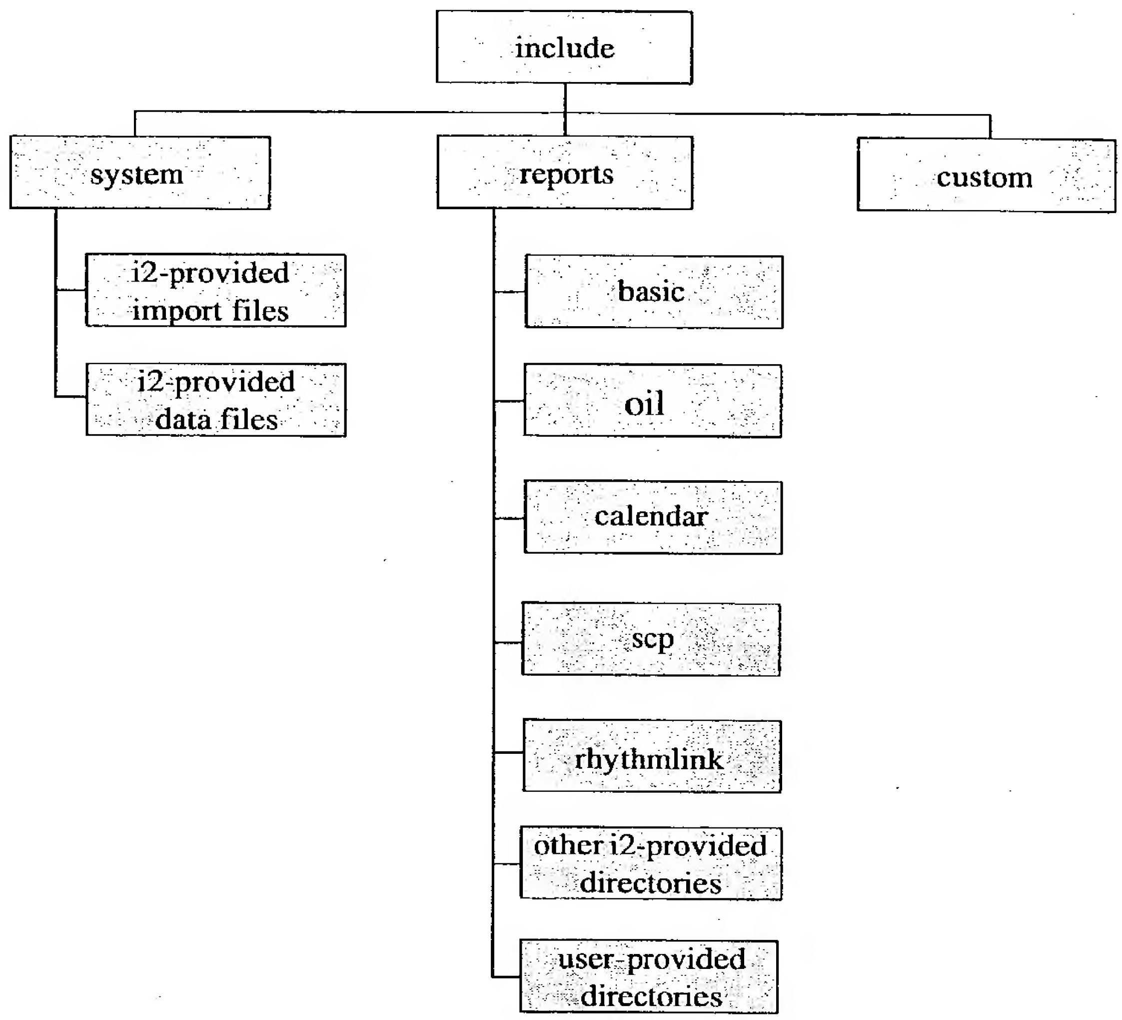
- *include* - the reference location (directory) for all input data to the engine. This directory is settable using the *include* option to the engine. Its default location is the location of the executables.
  - *custom* - the default location for the license files (*scp_engine.lic*) and startup customization (*scp_engine.opt* and *scp_ui.opt*).
  - *system* - the directory with model and user-specific customization settings. This directory is settable using the *system* option to the engine. Its default location is the system directory under the executables directory.
  - *reports* - the directory in which all reports information resides. Its default location is the reports directory under the executables directory. This directory is settable using the *reports* option to the engine.

The *system* directory contains i2-maintained files that you should **NOT** be editing during an implementation. We do not worry about this in the training examples. However, you generally should create your own *system* directory and arrange to have it processed by RHYTHM SCP after the standard *system* directory has been processed if you need to customize the *system* directory.

Likewise, do **NOT** edit the contents of the i2-provided *report* subdirectories. Make your own *report* subdirectory and arrange to have it override the standard *report* subdirectories.

FIGURE 2 shows the directory structure for *scp_engine*, *scp_ui*, and *scp_batch*.

FIGURE 2 System Setup



2.2.2 Environment Variables

Environment variables are a form of communication between the engine and the operating system of the machine the engine is running on. They can pass information to and from different processes, such as the engine and the UI, or a UI from one user to a UI of another user.

Information about the surrounding execution environment of the server and/or client is useful to have. RHYTHM SCP initializes environment variables that hold such information when the server or client is started. Examples of environment variables are:

- $I2_PID - the process id of the engine process
- $I2_APP - the application name
- $I2_HOME - the application directory (location of the executables)
- $I2_PORT - the port number currently being used
- $I2_REPORTS - the reports directory
- $I2_DATA - the data directory
- $I2_IMPORT - the default location of files for import
- $I2_EXPORT - the default location of files for export
- $I2_PRINT - the print command to be used for printing layouts and reports

2.2.2.1 Environment Functions

OIL functions for use with environment variables are *getenv* and *setenv*. The *getenv* function obtains the current setting of an environment variable. The *setenv* function sets the value of an environment variable. Formatting for these functions is shown below.

The *getenv* function is formatted as shown below:

```
getenv ("variable name")
```

where *getenv* must be typed as displayed and substitutions must be made for *variable name*. This expression returns the current value of this variable.

For example,

```
getenv ("$I2_IMPORT")
```

returns the current value of $I2_IMPORT.

The *setenv* function is formatted as shown below:

```
setenv ("variable name", "value")
```

where *setenv* must be typed as displayed and substitutions must be made for *variable name* and *value*. This expression sets the environment variable to the specified value.

For example,

```
setenv ("$I2_EXPORT", "/users/me")
```

sets the value of $I2_EXPORT to the directory */users/me*.

You can give an environment variable any name. Environment variables set in one environment are not necessarily accessible in all environments.

2.2.2.2 System Directory Files

Several key files reside in the *system* directory. These files exist prior to the receipt of RHYTHM SCP and usually only the *.dat* files require modification. They include the following files:

- user.imp
- user.dat
- meta_model.imp
- meta_model.dat
- measure_model.imp
- measure_model.dat

The *user.imp* and *user.dat* files contain user-specific customizations (e.g. user identities). *meta_model.imp* and *meta_model.dat* contain user-defined fields for extending models. *measure_base.imp* and *measure_base.dat* contain recognized units of measure and their associated conversions.

2.2.3 Defining User Models in user.dat

New users are defined in *user.dat*. If your user id is not in the global list of User Models at the server, then you can connect to the server, but only as the default New_User (if New_User is specified in *user.dat*). If New_User is not defined in *user.dat*, then anyone whose ID is not specified in *user.dat* cannot connect to the server. It should be noted that the person who initiates the engine is defined as the super-user. Therefore, to use the scp_ui `-user` option, the user must be the person who started the engine.

Additionally, a specification for the mix of user-specific custom and standard reports created in memory at startup, or incrementally, resides in *user.dat* for each User Model. Whether the mix is created at startup or incrementally is dependent upon the *preload* engine option. If the *preload* option is set to *true*, the application startup time is slow but the screens come up quickly. If the *preload* option is set to *false*, the application comes up faster but the initial display of each screen is slower. The benefit of setting *preload* to *true* is all reports are forced to be correct for the application to successfully start. **Note:** The default value of *preload* is *false*.

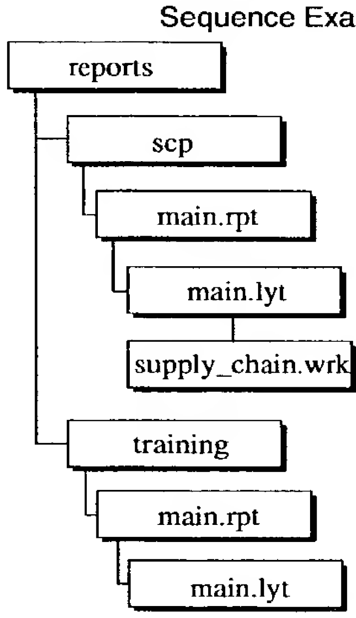
Another Note: You can put the options that you will want most often in a separate text file, then load this file with the *scp_engine* option *option_file my_opts*.

The User Model has several fields that must be defined when creating a user. These include:

- *name* - corresponds to NT or UNIX user IDs
- *report_directories* - directory with reports specific to the user being defined
- *organization* - pre-specified user whose customizations are to be included
- *include* - pre-specified user whose customizations are to be included for a particular report directory (*include* is a field in the report directory submodel of the User model)
- *sequence* - directories with smaller sequence numbers override those with larger sequence numbers (*sequence* is a field in the report directory submodel of the User model)

Each user can have its own set of report directories, although, user-specific customizations should be stored in a separate directory. Definitions for these reports and all their components are incremental, thus, the last one read wins (hence the *sequence* property in the User Model specification). For instance, in the example shown in FIGURE 3, the main window for user *teacher* is defined under the directory *training* and the main window for user *student* is defined under the directory *scp*. Both users are using the same *supply_chain.wrk* file under the directory *scp*.

FIGURE 3 Sequence Example



User	Reports Directory	Sequence
teacher	training	1000
teacher	scp	1200
student	scp	1200

FIGURE 4 and FIGURE 5 show sample *user.imp* and *user.dat* files.

FIGURE 4 user.imp Example

```
import_text_file user
{ file: "user.dat";
  delimiter: ":";
  worksheet () {
  this = users;
  [A1: name = #;]
  [B1: full_name = #;]
  [C1: organization = #;]
  [D1: report_directories.directory = #?".";]
  [E1: report_directories.include = #;]
  [F1: report_directories.sequence = make_type(#, integer) ?
  report_directories.count;]
  [G1: current_data_directory = option("data") ? (option("spec") ?".");]}
import_record: A1 B1 C1 D1 E1 F1;}
```

FIGURE 5 user.dat Example

```
# name:   full_name:  organization:  directory:   user:  sequence
#=================================================================
[unspecified]:  Default:     :              scp:          :      1000
[unspecified]:  Default:     :              calendar:  :       1400
[unspecified]:  Default:     :              rhythmlink::        1600
[unspecified]:  Default:     :              oil:          :      1800
[unspecified]:  Default:     :              basic:          :    2000
New_User:       New User:    :              [unspecified]::     [unspecified]
teacher:        teacher:     :              teacher:        :    800
teacher:        teacher:     :              scp:            :    1000
teacher:        teacher:     :              calendar:       :    1400
teacher:        teacher:     :              rhythmlink:     :    1600
teacher:        teacher:     :              oil:            :    1800
teacher:        teacher:     :              basic:          :    2000
```


Using the *user.dat* file shown, all users receive a copy of the contents of *New_User*. The default *New_User* gets its directories from the *[unspecified]* user. **Note:** If the *user.dat* is not completed properly, the client may be identified as *New_User* instead of the intended user (opening standard reports only, no customizations).

2.2.3.1 Defining Static Variables

OIL provides support for static-type variables that allow values to be set and retained between opening, closing, and reopening RHYTHM SCP for such things as filter and user preference dialogs. This is achieved through the user-defined fields on the User Model. An example use might be for controlling the display of time information. By adding a logical field to the user model (e.g. *time_info*), you could control the time stamping of your output.

```
if(user.time_info, echo("Starting export at" &
                now.string));
```

For more information on the User Model, refer to the *RHYTHM SCP Model Reference Manual.*

2.2.4 Defining Units of Measure in measure_base.dat

RHYTHM SCP supports defining domain-specific units of measure. Units of measure are defined in the *measure_base.imp* and *measure_base.dat* system files. New units of measure are defined in terms of the base units of measure. The base units of measure are:

- length
- mass
- current
- temperature
- time
- money
- pieces
- independent

Conversion factors are specified between various units of measure. FIGURE 6 and FIGURE 7 show sample *measure_base.imp* and *measure_base.dat* files.

FIGURE 6 measure_base.imp Example

```
import_text_file measure_base
{ file: "measure_base.dat";
  worksheet ()
{ this = measure_bases; //this model uses special syntax
  [A1: measure = #;] //to indicate multi-dimensional UOM
  [B1: name = #;]
  [C1: full_name = #;]
  [D1: factor = #;]}
import_record: A1 B1 C1 D1;}
//It is assumed that the base unit for time is in seconds
```

FIGURE 7 measure_base.dat Example

```
# measure                    name  full_name    factor
#=====================================================================
money                        dol   dollar       1
pieces                       pt    part         1
mass                         kg    kilogram     1
mass                         g     gram         .001 kg
mass                         mg    milligram    .001 g
length                       m     meter        1
length                       cm    centimeter   0.01 m
length[3]                    l     liter        1
length[3]                    ml    milliliter   .001 l
length*mass*time{-2]         N     Newton       1
length[2]*mass*time[-2]      J     Joule        1
time                         sec   second       1
time                         min   minute       60 sec
```

2.2.5 Defining User-Defined Fields in meta_model.dat

Most models are extendable with user-defined fields. User-defined fields are specified in *meta_model.dat* through the Model_Type Model. Every user-defined field is of a specific type. Type-specific formats are used for display and input. FIGURE 8 and FIGURE 9 show sample *meta_model.imp* and *meta_model.dat* files.

FIGURE 8 meta_model.imp Example

```
import_text_file meta_model
{delimiter: ";";
  file: "meta_model.dat";
  worksheet ()
{ this = model_types;
  [A1: name = #;] /
  [B1: fields.type = #;]
  [C1: fields.name = #;]
  [D1: fields.description = #;]}
import_record: A1 B1 C1 D1;}
```

FIGURE 9 meta_model.dat Example

```
# Model;  Value_Type;   Field_Name;   Description
#=========================================================================
Buffer;   Symbol;       short_name;
Item;     Symbol;       short_name;
```

The *meta_model.dat* defines a field *short_name* for the *Buffer* and *Item* models. Such fields are really understood only at the client, but the software can perform general calculations with them by using them in OIL expressions.

2.2.6 Reports Directory

The *reports* directory contains all report information. Its location defaults to the *reports* directory under the executables directory. However, its location is settable using the *reports* option.

All standard reports and utilities reside in the following subdirectories under the *reports* directory:

- *basic* - default formats, styles, and reports
- *oil* - reports for the OIL interactive editors
- *rhythmlink* - reports for RhythmLink
- *calendar* - reports for the Calendar Editors
- *scp* - standard SCP reports

All custom reports reside in other subdirectories under the *reports* directory.

There are advantages and disadvantages to having the *reports* directory. One advantage is incremental changes can be made for a specific user without affecting the customization for others. One disadvantage is the dependencies between reports and their components can become complicated and need to be well documented.

2.2.7 Using .opt Files

Options and flags are generally set to the *scp_engine* and *scp_ui* in-line. These options and flags can be placed in *.opt* files rather than specifying them in-line. When invoked, *scp_engine* by default looks for a file called *scp_engine.opt* in the same directory, and *scp_ui* looks for a file called *scp_ui.opt*.

The *startup:* option for *scp_engine* specifies an OIL expression to be run at startup. Since certain OIL functions can specify the use of other files, any amount of work can be accomplished during startup, including importing orders, forecasts, planning and running strategies, etc.

2.2.8 Using .in Files

.in files are OIL script files which are interpreted and executed by *scp_engine*. They contain a series of OIL expressions and OIL functions and are usually invoked by a *do_file* function (i.e. `do_file("my_commands.in")`).

The *before_import.in*, *after_import.in*, *before_export.in*, and *after_export.in* files are automatically invoked by the *import* and *export* functions. If the first parameter of the *import* or *export* function is a directory and that directory contains a *before_import.in*, etc., file, the contents of this file are evaluated before or after the other files in the directory are imported or exported.

2.3 More Basics

As mentioned previously, OIL is a strongly *typed* language. It consists of numerous models that represent the many elements of a supply chain. Each model has submodels, fields, extensions, and functions, and each submodel has its own fields, extensions, and functions.

2.3.1 OIL Models

A model is a data structure in OIL that:

- represents an entity in the supply chain world
- has fields that can either produce values or execute commands
- has at least one field used for identification purposes, usually called a *key* field
- has another model that is the designated *owner*

Models are the building blocks for the various entities in the supply chain. They contain all the necessary information that characterizes the entity being modeled. Models also form the framework within which the behavior of individual entities can be extended for specific requirements.

Every model is also a *type* (Model_Type). Each model has a key field that allows RHYTHM SCP to distinguish individual instances from one another (e.g. used by the *find* function). Each model has submodels from which the parent model can be accessed using the *owner* field under the submodel. For example, consider the following expressions:

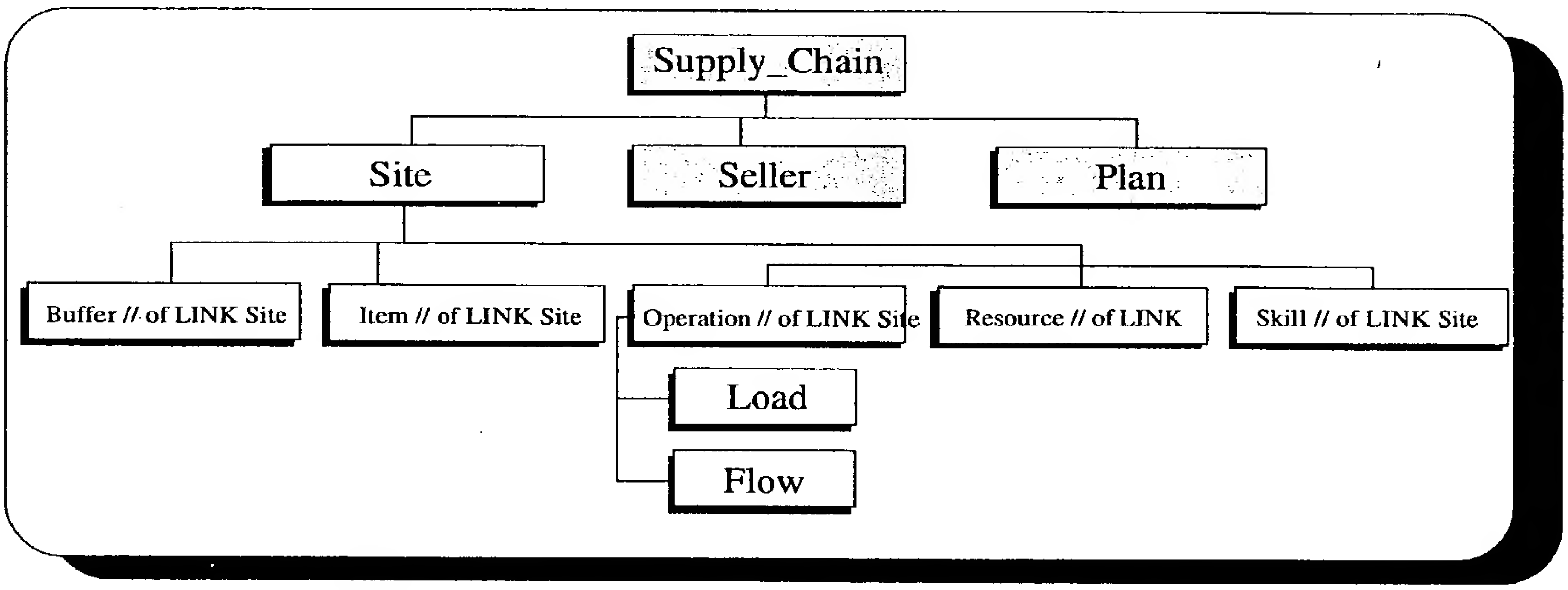
```
load.owner

operation.owner
```

where *load* and *operation* are the submodels. The first expression returns *Operation* as the parent model. The second expression returns *Site* as the parent model.

FIGURE 10 shows the relationship between the *Supply_Chain*, *Site*, *Seller*, and *Plan* models in a supply chain. (The Site model is expanded in this example.)

FIGURE 10 Models and Submodels



2.3.2 Model Fields

The following types of fields exist:

- standard
  - values
  - commands
- extension
- user-defined
- key
- owner

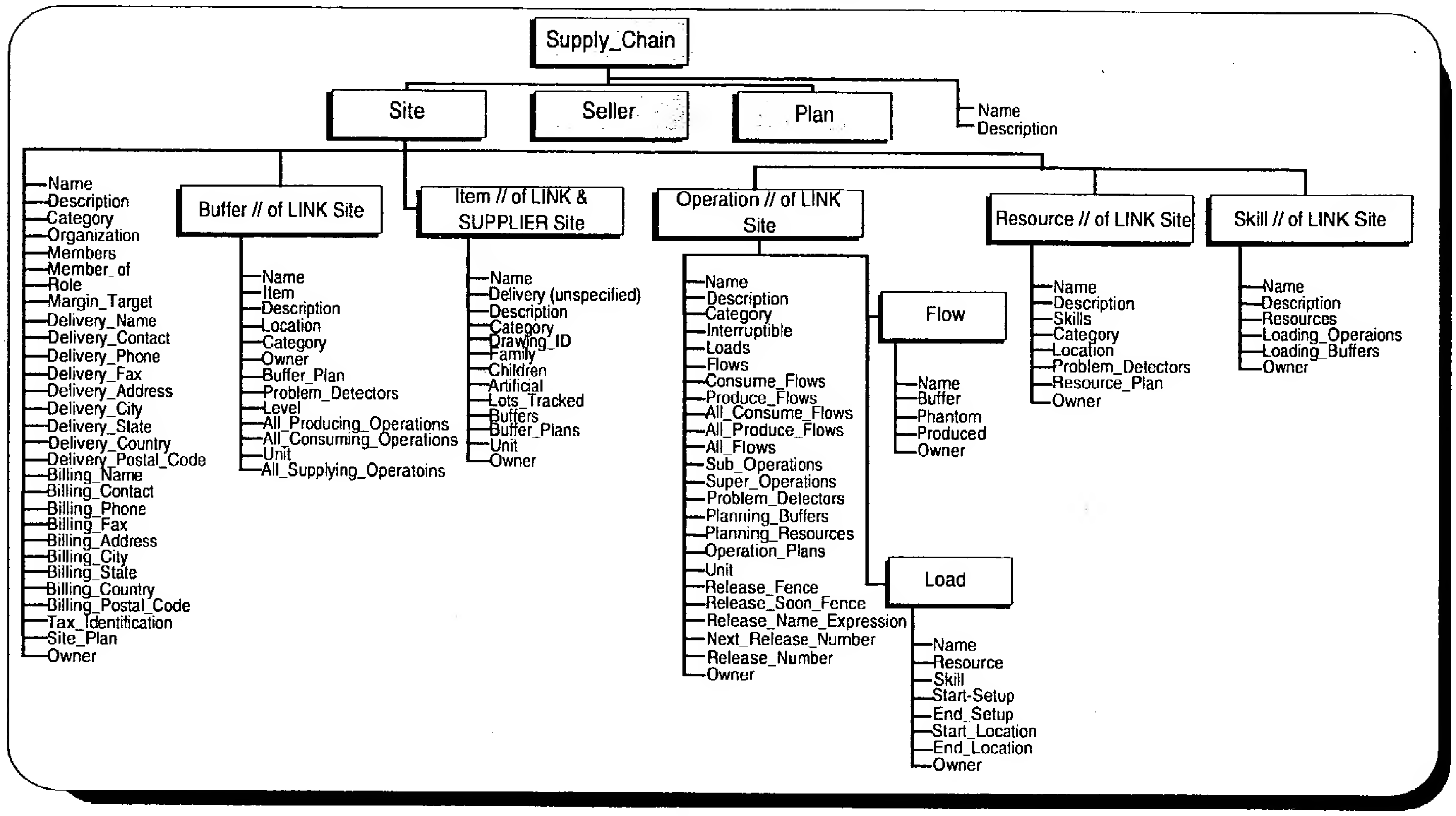
Fields correspond to data elements of specific types within a model. Accessing a field within a model is equivalent to invoking a *function* with the same name as the field that has the model as an argument. For example, the following expressions are equivalent:

```
Site.name    <=>        name(Site)
```

where *Site* is the model and *name* is the *field*.

FIGURE 11 displays the fields of the Supply_Chain and Site models.

FIGURE 11 Supply_Chain and Site Model Hierarchy



Fields describe model-specific information (e.g. *name* field of the *Site* model). Each field is a function in OIL. They can be defined to take more than one argument. If a field is not defined as *export_only*, it may have a built-in function for setting its value (named as "*set_xxx*").

For example, the expression

```
buffer_plan.on_hand()
```

returns the quantity on hand at the buffer. To set that quantity, write the following expression:

```
buffer_plan.set_on_hand(10)
```

Many fields have default values. The exception is a model's *key field* which does not have a default value. The value *[unspecified]* (built-in identifier) is typically used for default values whose *type* is a model.

Several fields name extensions. These extensions are used to customize the behavior of the particular model. The extensions identify which submodels and fields are active in a model. For example, *Site.role = "LINK"* declares the role of this submodel Site as a link in the supply chain. As such, the site can be modeled in detail and is used to transfer items between buffers, and to place requests and provide promises to other Sites.

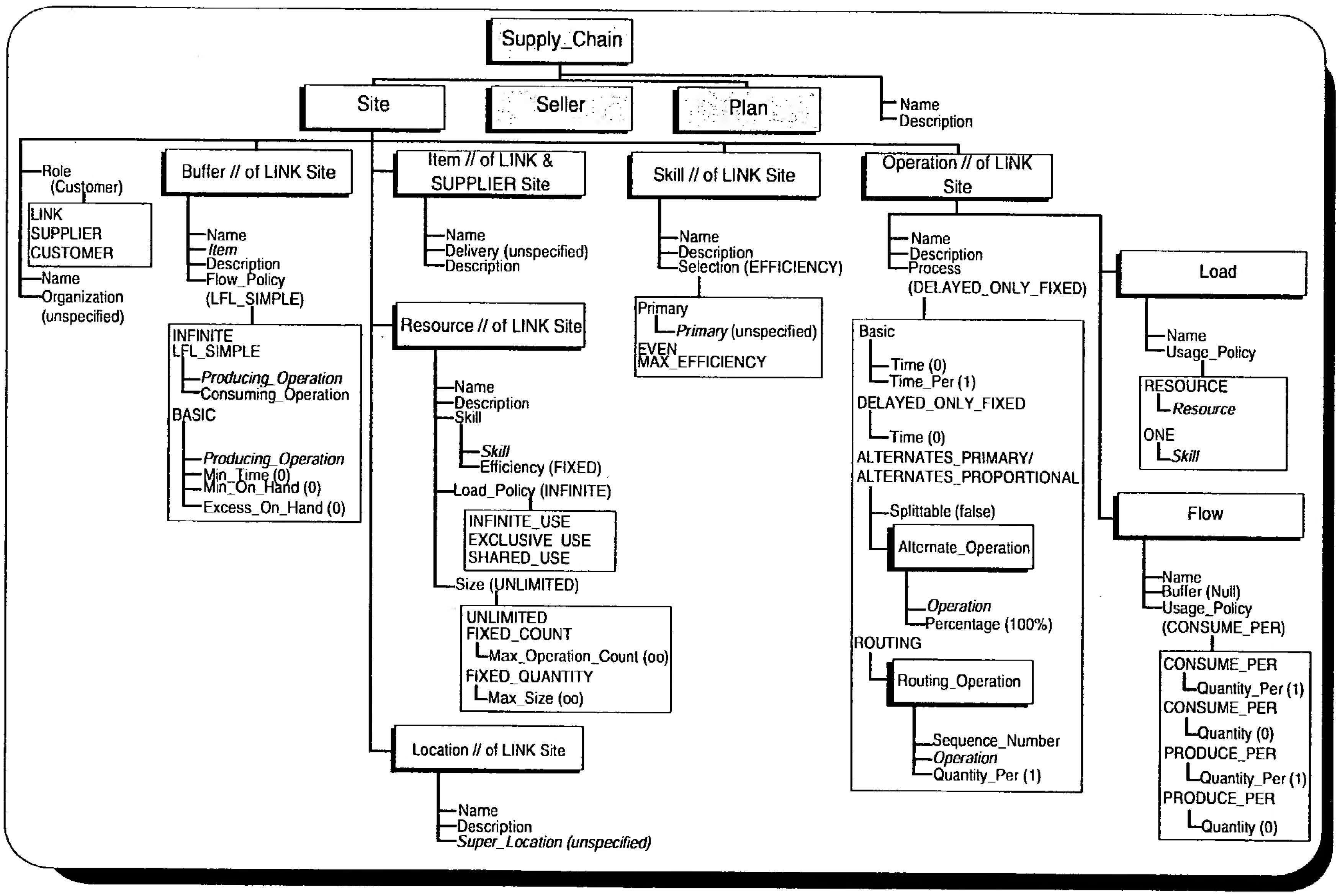
Typically, the value of an extension selector such as "LINK" affects whether certain fields of a model are active. The *exists( )* function tests whether or not a field is active. For example, the expression `exists(operation)` returns "TRUE" if an operation field exists and "FALSE" if an operation field does not exist. When a field is not active, invoking it causes the function *nonexistent* to initiate, which has the practical effect of a null result. This is **not** an error.

2.3.2.1 Model Extensions

RHYTHM SCP uses *extensions* in conjunction with *fields* to customize the behavior of a model. The *extension*, in effect, adds *fields* to the model. Refer to the *RHYTHM SCP Model Reference Manual* for more information on *extensions*.

FIGURE 12 shows the *extensions* for certain *fields* within the *Site* model.

FIGURE 12 Extensions of the Site Model



2.3.3 Model Functions

Functions operate on one or many arguments to return a result of a specified *type*. They can be nested to as many levels as there are fields defined in a model or submodel. For example, the following expressions *A* and *B* are functionally equivalent.

```
start(
   horizon(
      find(
         plans(
            find(supply_chains,
            "My Supply Chain")
         ), "Actual Plan")
      )
   )
```

Expression A

```
supply_chains.find("My Supply Chain")
   .plans.find("Actual Plan")
      .horizon
         .start
```

Expression B

A function that expects multiple arguments can also be written as a field or function of the first argument. For instance, consider the following expressions:

```
problems(Site_Plan, Problem_Category)

Site_Plan.problems(Problem_Category)
```

where the first expression is a *function* using *problems*, which is a field of the *Site_Plan* model, as written in the second expression.

2.3.3.1 Using the nonexistent Function

Most functions do nothing at all when encountering *nonexistent*. For example, the expression `f (E)` returns nothing if *E* encounters *nonexistent*. The only exceptions are these OIL functions and operators:

- *exists* - this function returns FALSE if the second parameter is `nonexistent`; otherwise, it returns TRUE
- == - these operators return FALSE if one or both parameters are `nonexistent`
- *!=* - these operators return TRUE if one or both parameters are `nonexistent`
- *contains* - this function returns FALSE if its second parameter is `nonexistent`
- *find_or_nonexistent* - this function returns `nonexistent` when no matches are found
- *found* - this function returns FALSE if its second parameter is `nonexistent`
- *?* - this operator returns the first parameter if it does not abort out with `nonexistent`; otherwise, it returns the third parameter

The expression `exists (E)` returns “FALSE” if *E* encounters *nonexistent*, otherwise, it returns “TRUE”. The expression `E ? X` returns *E* if it does not encounter *nonexistent*, otherwise, it returns *X*.

The following is an example expression using the *nonexistent* function:

```
[G1 request = if(deliv_op.exists,
deliv_op.motive_request, nonexistent);]
```

Note: This expression is redundant and can be simplified to be more efficient:

```
[G1 request = deliv_op.motive_request;]
```

The *nonexistent* function is the OIL equivalent of "undefined". You can give undefined things this value so that you can test for it with other functions.

You can create something with the value *nonexistent* using the *make_type* function:

```
variable old_request = make_type(nonexistent, Request);

if(old_request.exists, echo("There is already a request"),
set_variable(old_request, new_request))
```

2.3.3.2 Symbolic Constants vs. Zero-Parameter OIL Functions

Zero-parameter functions require no parameters and do not require quotes in their syntax. *nonexistent* is actually an OIL function that takes no parameters. Other common *zero-parameter functions* include:

- *now* - returns current date and time
- *rand* - returns an evenly distributed random number between 0 and 1
- *report* - returns the current report
- *user* - returns the user for the client, or nonexistent
- *close* - closes the current report window

In contrast, *symbolic constants* (such as the various model extension names) are predefined values within OIL that have specific meanings. They are not *functions* and need to be placed in quotes in OIL expressions (e.g. *my_site.role = ="LINK"*). However, it is an error to quote *zero-parameter functions* (e.g. *nonexistent*).

2.3.4 Types and Conversions

Values have *types* that are dependent on the OIL expressions. For example, consider the following expressions:

```
plans.first.horizon

plans.first.horizon.start
```

The first expression returns a *type* of *Date_Range*. The second expression returns a *type* of *Date*. OIL consists of the following types, grouped in the table below by category:

Category	Types
Numeric	Integer, Number, Percentage
String Symbol	ID, Logical, Pathname, String, Symbol
Date Time	Date, Date_Range, Horizon_Date, Restriction
Quantity Measure	Measure, Measure_Unit, Money, Quantity, Quantity_Range, Time
Special	Cell_State, Change_Category, Event, Expression, Predicate, Priority, Reference, Type, Typed_Value, Void
List	Homogenous, Keyed, Bucketed, Recursed

Refer to the *RHYTHM SCP Model Reference Manual* for detailed definitions of all the types.

Any conversion required from one type to another is managed intelligently (e.g. *String* to most other *types* is converted automatically). **Note:** Any *type* to *String* is not automatic. Use the *string ( )* function to convert a *Symbol*, etc. to *String*.

The following expression shows an example of *types*:

```
operation.flows.first.quantity_per = "23"
```

quantity_per is of *type Quantity* but *"23"* is of *type String*. Although this expression will work without explicit *type* conversions, a *type conversion function* can be used to force the conversion. The *string ( )* function is an example of a type conversion function. Refer to the *RHYTHM SCP Model Reference Manual* for definitions of all the type conversion functions.

2.4 Engine and UI Options

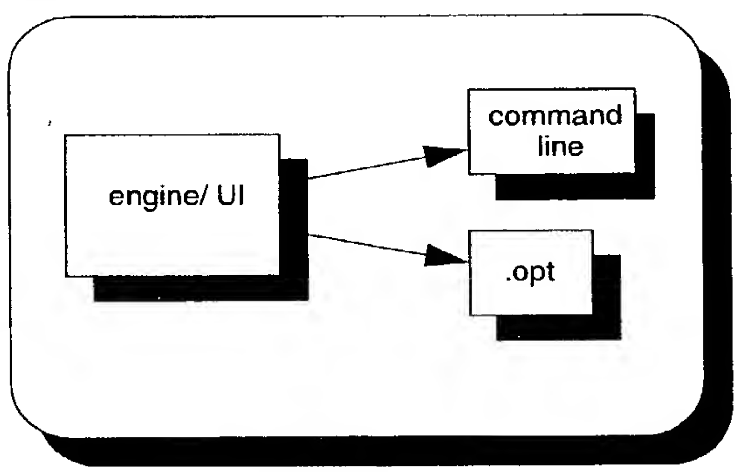
RHYTHM SCP is designed to be customizable. Customization may be implemented through one of the following mechanisms:

- *engine* - Rhythm SCP options supplied at the engine level (See FIGURE 13)
- *UI* - Rhythm SCP user interface (UI) options supplied at the client level (See FIGURE 13)
- *reports, layouts, worksheets, styles, formats*
- *data files*

RHYTHM SCP option *(.opt)* files allow for customization of such items as the following:

- behavior of the engine
- behavior of the UIs

FIGURE 13 Engine/Client Option Architecture



2.4.1 Overview

This section begins with a brief overview of the option mechanisms. It then presents a discussion of topics which apply to both the engine and UI and to RHYTHM SCP options:

- Naming Option Files
- Specifying Options

The remaining sections discuss the following topics:

- All available RHYTHM SCP engine options and UI options are listed and described
- Environment variables

2.4.1.1 Engine

The *engine* program runs with a specific set of options. The option values with which the program runs may be modified by supplying:

- RHYTHM SCP option/value pairs on the RHYTHM SCP engine command line
- RHYTHM SCP option/value pairs in *.opt* files

2.4.1.2 UI

The *client* program runs with its own specific set of UI options. The option values with which the program runs may be modified by supplying:

- RHYTHM SCP option/value pairs on the RHYTHM SCP client command line
- RHYTHM SCP option/value pairs in *.opt* files

2.4.2 Naming Option Files

The RHYTHM SCP application options can be specified for either all RHYTHM SCP applications or specific RHYTHM SCP applications, and can be specified for an entire customer site or for particular users.

2.4.2.1 Naming Rules

The following rules govern the names that option files should be given:

- Options for RHYTHM SCP applications have the suffix *.opt*.
- Options that apply to *all* RHYTHM SCP applications should be specified in files named *i2.opt*.
- Options for the engine level (the file containing RHYTHM SCP application options for the engine level) should be named *scp_engine.opt*.
- Options for the UI level (the file containing RHYTHM SCP options for the UI level) should be named *scp_ui.opt*.
- Option files which are *Site-wide* should be placed in the custom subdirectory of the RHYTHM SCP directory. For example, if RHYTHM SCP is installed in the directory */usr/local/Rhythm*, then the site-wide options would be in the directory */usr/local/Rhythm/custom*.
- Option files which are *User-specific* should appear in the user's home directory.
- Option files can have any name you wish as long as you use the *option_file* option to name them

2.4.2.2 Search Rules

The following rules explain the directories that are searched for option files. To obtain the values for options, RHYTHM SCP searches a sequence of directories (called a search path) in a specific order. The value of an option found in a file in one directory overrides, or resets, the value of the same option found in a file in any of the directories in lower priority in the search path. This search sequence allows a user to override the site-wide options, which can override the i2 Technologies-supplied options:

- *Command line options* - highest level of priority. Any options specified on the command line call of a RHYTHM SCP application override any of the options specified in the files described below.
- *Standard RHYTHM SCP settings* - i2 Technologies-supplied defaults.
- At the same level (user-specific, site-wide, standard), *application specific* settings will override all RHYTHM SCP settings.

2.4.2.3 Default List of Option Files

The default list of option files to read comes from the *app_name_options* environment variable. If this variable does not exist, then RHYTHM SCP searches each of the option files shown in Table 2 for the RHYTHM SCP options, if they exist. The following conventions are used:

- *<app_dir>* indicates the pathname to the executable file which runs the RHYTHM SCP program.
- *<app_name>* indicates the RHYTHM SCP application program (which may be overridden with the *name* command line option) that is being run. For example, *scp_engine*.
- $HOME indicates the user's home directory.

The table is ordered from highest (most specific; current directory) to lowest (most general; home directory) level of override priority. Command line options override option file values.

Table 2: Search Rules for RHYTHM SCP Options

Rule	File	Level	Scope
9	command line options	user specific	all RHYTHM SCP
8	*<app_name>*.opt	site-wide	application specific
7	i2.opt	site-wide	all RHYTHM SCP
6	*<app_directory>*/*<app_name>*.opt	standard RHYTHM SCP	application specific
5	*<app_directory>*/i2.opt	standard RHYTHM SCP	all RHYTHM SCP; should not be altered by customers
4	*<app_directory>*/custom/ *<app_name>*.opt	site-wide	application specific; may be altered by customers
3	*<app_directory>*/custom/i2.opt	site-wide	all RHYTHM SCP; may be altered by customers
2	$HOME/*<app_name>*.opt	user specific	application specific
1	$HOME/i2.opt	user specific	all RHYTHM SCP

2.4.3 Specifying Options

Each option has a name and a data type. The data type is a descriptor for how the value of the option is to be entered on the command line or in the option file. The available data types are as follows:

- *Boolean*- a Boolean flag which has a value of either TRUE or FALSE.

  As a command line option, this type may be implemented by preceding the option with an optional plus or minus sign to indicate its value:

+option	set it to a value of TRUE
-option	set it to a value of FALSE
option	toggle the TRUE/FALSE value of the option

  It may also be implemented by preceding the option with a minus sign, and entering an explicit TRUE or FALSE (case insensitive) following the option name.

  Examples:

appname	*+turn_me_on*	*-turn_me_off*	*toggle_me*
appname	*-turn_me_on TRUE*	*-turn_me_off FALSE*	

- *Float* - the option represents a floating point number.
- *Function* - Looks like some other data type to the user, usually *String*.
- *Integer* - the option represents an integer number.
- *String* - the option represents a character string.

2.4.3.1 Command Line Option

Option values may be specified on the command line. For example,

scp_engine port 6162 &

where

scp_engine	is the name of the program
port	is an integer option set to integer 6162
&	run program in the background

If an option on the command line is not recognized, a help message is displayed, and the program terminates.

2.4.3.2 Option File Format

The option (*.opt*) file format is modeled after X11 resource files, except that no wild-carding is performed. There is one line per option. Each line has the option name followed by a colon, then by a value. For Booleans, the value will be either TRUE or FALSE:

```
name: value
```

A space may follow the colon but not precede it.

An example option file:

```
! Comments begin with !
! The following line initializes the open option with the string
! my_customer.i2
open: my_customer.i2
! Upon engine startup, evaluate an OIL expression that plans to
! satisfy all requests in all the models in the engine
startup:
supply_chains.for_each(#.plans.for_each(#.site_plans.for_each
   (#.plan_to_satisfy_all_requests)))
```

2.4.4 Available Options

To view a list of the available RHYTHM SCP options which can be used as command line options, the executable should be run with the *help* option. To view engine or UI options, that users implement frequently, e.g. *data, host, open, port*:

scp_engine	*help*
scp_ui	*help*
scp_batch	*help*

To view all UI options (UI options that users implement frequently, and options that are used to customize RHYTHM SCP, e.g. *default_format, default_style*):

scp_engine	*help all*
scp_ui	*help all*
scp_batch	*help all*

To view detailed help on one particular option:

scp_engine	*help <option>*
scp_ui	*help <option>*
scp_batch	*help <option>*

If an option on the command line is not recognized, a help message is displayed, and the program terminates.

Note: At the time of printing this manual, due to a limitation in Windows NT, this capability does not work correctly for *scp_engine* and *scp_ui* executables. It does work correctly in Unix.

2.4.4.1 Standard Options

Table 3 lists the engine and UI options that users implement frequently and that are common to both the engine and UI executables (*scp_engine help*, *scp_ui help* **and** *scp_batch help*). Table 4 lists the options that users implement frequently with the engine executable but that do not apply to the UI executable (*scp_engine help* only). Table 5 lists the options that users implement frequently with the UI executable but that do not apply to the engine executable (*scp_ui help* and *scp_batch help* only).

Table 3: Standard Options - Engine and UI

Option	Type	Default	Definition
deadman_timer	U-Integer	FALSE	Kill engine when idle more than the indicated number of MINUTES. Zero (default) disables this option.
debug	Boolean	FALSE	Must be set to TRUE (1) to enable OIL debugger breakpoints. This option is automatically set to TRUE when it encounters the first enabled breakpoint.
default_metrics_percentage	U-Integer	80	Only report users of the top X% of memory.
help	Function		Print documentation for all options. Specifies all of the known option names, types, and values.
host	String	"localhost"	Default host name. If empty, use the current machine name.
identifier_compatibility	Boolean	FALSE	If TRUE, identifiers will be converted to strings (or symbols) with a warning.
ignore_strings	String	"."	Ignore strings from this file.
include	Function	(NULL)	Adds a directory to the list used for opening data files. If a relative pathname is given for a data file or directory, the *include* path is searched to find it. The default is "." (the current directory). *include* may be specified multiple times. Each time, the specified path is prepended to the previous path. *path* may be a single directory, or a list of directories delimited by one or more " \t,:" (**Note:** report/layout/worksheet/style/format files use the search path defined by the *user* database).
java_src	Boolean	FALSE	Parse java source, generating strings.
language	Function	(NULL)	Default language.

Table 3: Standard Options - Engine and UI

Option	Type	Default	Definition
log_file	String	"log_file"	This command line option saves output to the named file in addition to writing to stdout (or stderr). This option works with any executable (scp_engine, scp_ui, rl_engine, modelgen, etc.). Output is flushed to the file after every line. The format for this option is: scp_engine port 0 log_file foo
max_clients	Integer	5	Maximum number of clients allowed to connect to this engine.
max_errors	Integer	10	File reading error messages will be printed at most *max_errors* times per file.
max_records	U-Integer	1000	Maximum number of records to be viewed per table by the GUI.
mprof	Function	0	Report profile data when total allocation changes by N megabytes.
mprof_log	Function	./memory.mprof	Record memory profiling file in the specified log file.
mprof_metrics	Function	0	Report metrics every X megabytes.
open	String	""	Restore data from this directory. To prevent importing on top of a restored dataset, make sure that the following entry does not exist in the *i2.opt* or *scp_engine.opt* files: *data: <directory>* Select the *Import* and *Save* (*Save As*) options in the *File* menu of the main window to restore and save the model.

Table 3: Standard Options - Engine and UI

<table>
<tr><th>Option</th><th>Type</th><th>Default</th><th>Definition</th></tr>
<tr><td>option</td><td>Boolean</td><td>FALSE</td><td>If set to TRUE, report where options originate. A trace of which options come from what option files is printed.</td></tr>
<tr><td colspan="4">Example of engine options reported when option is TRUE:

----- Reading options from ./i2.opt -----
Option include = tests . /rhythm/data
----- Reading options from ./scp_engine.opt -----
Option open = electronix.i2
Option startup =
supply_chains.for_each(#.plans.for_each(#.site_plans.for_each(#.plan_to_satisfy_all_requests)))
Option command_execute = true
Option print_usage = mean sum realtime, mean sum memory, mean run pause
Option recurse_depth = 100
Restoring scp_engine version 3_07 A on zip started by john at 03/14/98 15:47:13
supply_chains.for_each(#.plans.for_each(#.site_plans.for_each(#.plan_to_satisfy_all_requests)))
Handling requests from port 27111</td></tr>
<tr><td colspan="4">Example of UI options reported when option is TRUE:

----- Reading options from ./i2.opt -----
Option include = tests . /rhythm/data
----- Reading options from ./scp_ui.opt -----
Option splash = false
Option user = electronix</td></tr>
<tr><td>option_file</td><td>Function</td><td>(NULL)</td><td>Read an options file.</td></tr>
<tr><td>port</td><td>Integer</td><td>27111</td><td>TCP Port of engine.</td></tr>
<tr><td>preload</td><td>Boolean</td><td>FALSE</td><td>If set to TRUE, preload all reports, layouts, worksheets, styles, and formats.
If set to FALSE (default), report files are loaded incrementally. user <name> then causes scp_ui to use the <name> user.
In effect, the following entries are equivalent:
scp_engine -preload user <name>&; scp_ui
scp_engine -preload&; scp_ui user <name>
The following entries are also equivalent:
scp_engine -preload user <name>
scp_engine -preload new_user <name>
If a value of FALSE is used on the engine, then the client comes up faster because it loads the reports only when necessary.</td></tr>
<tr><td>show_progress</td><td>Boolean</td><td>TRUE</td><td>If set to TRUE, display progress messages.</td></tr>
</table>

Table 3: Standard Options - Engine and UI

Option	Type	Default	Definition
startup_hook	String	"(NULL)"	Shell command to run after connecting to client or engine.
tdir	String	"."	Directory where the engine looks for translations.
term_width	Integer	80	Terminal width in characters. Used to widen the display output when the help feature documentation is truncated. A value of 132 is usually sufficient.
version	Function	(NULL)	Display the version number and date without loading data files. Using port 0 provides the same results: *scp_engine version 3_07 A of 04/01/98* *scp_ui version 3_07 A of 04/01/98*
which_server	Boolean	TRUE	Detects another server on the same port. Set to FALSE to avoid Rhythm2 server crashes when a Rhythm3 server is using the same TCP port as a Rhythm2 server. Set to TRUE to enable the feature of reporting which *scp_engine* is using the requested TCP port. Default is TRUE because few users will be using both Rhythm2 and Rhythm3 at the same time, and because you would have to work at it to use the same port.

Table 4 lists the options that users implement frequently with the engine executable but that do not apply to the UI executable (*scp_engine help* only).

Table 4: Standard Options - Engine Only

Option	Type	Default	Definition
cal_plan_period	Function	(NULL)	Specifies the default plan period for Calendar_Plan.
data	String	"(NULL)"	Import data from this directory. To prevent importing on top of a restored dataset, make sure that the following entry does not exist in the *i2.opt* or *scp_engine.opt* files: *data: <directory>* Select *Import* and *Save* (*Save As*) options in *File* menu of the main window to restore and save the model.
default_temp_table_threshold	Integer	2147483647	Specifies the maximum number of temp table records to keep in memory.
dts	Integer	FALSE	If set to TRUE, save all data and options to a directory named with the dts number.
dts_old	Integer	0	Save all data and options to /tips/data/dts/9999 (where 9999 is the dts number).

Table 4: Standard Options - Engine Only

Option	Type	Default	Definition
flow_policy_threshold	Float	1e-05	Buffer roundoff threshold value. Tells Buffers how close two numbers have to be to consider them equal (cannot depend on perfect equality because of the limitations of computers in dealing with real numbers). For example, if a Buffer requests a supplying operation for 10 units, and it comes back with 9.99 units, that is probably OK. But if it comes back with 9.5 units, that is probably not. By using a *flow_policy_threshold* greater than 0.01, but less than 0.5, Buffer code will interpret those numbers correctly. The default *flow_policy_threshold* is 0.00001. It demands a close match by default, so as not to throw away real material thinking it is roundoff. Math processors only give a fixed range of precision. This means that the biggest Flow_Plan in or out of a Buffer needs to be no more than 160,000,000 times as large as the *flow_policy_threshold.* (That is not something that can be controlled. It is a feature of the hardware.) With the default threshold, problems may begin to occur when dealing with Flow_Plans bigger than 1,600 (= 160,000,000 * 0.00001) units. Those problems can show up in a variety of ways, but the most common is a sudden explosion of tiny Operation_Plans, each just barely above the *flow_policy_threshold.* If you need to deal with Operation_Plans bigger than 1,600 units, and you do not need to accurately represent quantities as small as 0.00001, then you will probably want to set a higher *flow_policy_threshold,* but keep in mind the following rules: • Never set it higher than 1/4 the size of the smallest units you want to represent. For example, if the smallest units in your factory are a half of something, never set it above 0.125. • Never set it to a negative number. This causes the engine to run indefinitely. • Never set it greater than 1.0.
interaction_coefficient_destroyed_factor	Float	1	The interaction factor of destroyed problems.

Table 4: Standard Options - Engine Only

Option	Type	Default	Definition
interaction_coefficients	Boolean	TRUE	Activates interaction coefficients. When FALSE, forces all coefficients to be 1.0. *Interaction coefficient* is the measure of the likely propagation of problems from a FLO object (based on past experience). *Interaction coefficients* are learned as problem solving progresses. They reside on FLO objects and measure past experience with resolving problems on that object. If past resolutions have caused many problems (or dramatically changed the goal function), then this is reflected in the *interaction coefficient.* Each time a problem is resolved on a FLO object, a new *interaction coefficient* term is generated which measures the effect of that resolution. The actual *interaction coefficient* for the object is a weighted average of the terms.
license	String	"(NULL)"	Enter license key from the command line.
mt	U-Integer	4	Maximum number of commands that can be run in parallel.
mt_stack_size		10 MB	
new_eff_alt_behavior	Boolean	FALSE	When TRUE, use alternate Effective_Calendar operations only when the USE_EFFECTIVE_ALTERNATES change category is specified.
ofl_bucketize	Boolean	FALSE	Makes OFL problems bucketized. **Note:** Problems can extend more than a bucket in length. If set to TRUE, OFL problems will be no longer than one bucket in length.
ohc_split	Boolean	TRUE	Causes the move_in and move_out OHC resolves to split if necessary. Also splits operation plans (if necessary) while resolving overloads in OHC and FLC buffers using alternate_operations.
prob_selection_age_usage	Float	0	When nonzero, young problem ages are preferred when selecting problems. When you run a strategy, problems are created. These problems are dated (using the Date Activated field). When you make changes to the plan, new problems are created. When you run the strategy again, you can use *prob_selection_age_usage* to specify that the strategy looks only at the newest problems, and ignores older ones.
random_seed	Boolean	FALSE	Causes SDP to reset the seed to a random value.

Table 4: Standard Options - Engine Only

Option	Type	Default	Definition
rap_threshold	Float	0.0001	Request and promise quantity problem detection threshold.
reports	String	"$I2_REPORTS"	All relative User report_directories pathnames are relative to this path.
resource_ignore_tabu	Boolean	FALSE	When TRUE, resource resolvers ignore tabu restrictions.
save32	Boolean	FALSE	Save 64-bit binary files in 32-bit format.
spec	String	"(NULL)"	Read import files from this directory.
startup	String	"(NULL)"	Evaluate a worksheet expression at startup.
startup_file	String	"(NULL)"	Evaluate this expression file at startup.
system	String	"(NULL)"	System data directory. Import data from specified directory. The default is the system directory which contains data shipped with the system. Any or all of these files can be edited and placed in another directory (see data). The files include: *measure_base.dat* - Defines default units of measure. *meta_model.dat* - Defines user defined data fields. *user.dat* - Defines the user model, which tells *scp_engine* where to load report files. See separate section titled Environment Variables for more details.
tabu_enforce	Float	0.5	Adjusts tabu enforcement. 1 = always enforce. 0 = never enforce. 0.5 = default.
tabu_restrictions	Boolean	TRUE	Specifies whether SDP tabu restrictions are active. **Note:** If changed after startup, old restrictions are not flushed.
user	String	"(NULL)"	Preload all of the reports / layouts / worksheets / styles / formats for this user before servicing GUI requests. This is a name in the user *name* field in the *user.dat* file (User model).
user_data	String	"(NULL)"	User import directory.
user_spec	String	"(NULL)"	Read user import files from this directory.

Table 5 lists the options that users implement frequently with the UI executable but that do not apply to the engine executable (*scp_ui help* and *scp_batch help* only).

Table 5: Standard Options - UI Only

Option	Type	Default	Definition
allow_window_growth	Boolean	FALSE	If set to TRUE, report windows are always allowed to grow to fit currently visible layouts.
batch	String	"(NULL)"	An expression to execute INSTEAD OF displaying any windows: Example 1: *scp_ui batch "echo(3+5)"* Example 2: To save the current plan to a file with a filename containing the date and time that the file is saved (now): *scp_batch batch 'save("scp_" & now & ".i2")'* The expression could be placed in a file, e.g. *file.in*, and executed either as a parameter to the engine: *scp_engine startup 'do_file("file.in")'* or as a batch client: *scp_ui batch 'do_file("file.in")'* All batch commands start off executing on the engine. Formerly, they executed on the GUI (unless the GUI just sent them right back to the engine). This saves one or two round trips between the server and client. Batch commands do not load the current user's reports unless the *load_all* option is in effect.
batch_file	String	"(NULL)"	An engine visible file to execute INSTEAD OF displaying any windows.
display	Function		X display name.
doc_dir	String	"doc"	Directory where documentation files exist.

Table 5: Standard Options - UI Only

Option	Type	Default	Definition
initialize	String	"display_report(\"main\")"	Expression to display initial report window. *display_report* specifies the report that will be displayed when the *scp_ui* runs. An *open_report* command is sent to the engine, which evaluates the *display_report* expression. *display_report* is always compiled on the engine because it needs to read a new report.
laf	String	"windows"	Look and feel for the user interface. Value must be one of *windows* or *motif.*
max_initial_height	Integer	1000	Maximum initial height allowed for report window.
maximum_initial_width	Integer	1000	Maximum initial width allowed for report window.
popup_messages	Boolean	TRUE	If set to TRUE, display messages in popup window. If set to FALSE, display messages to the terminal.
resize	Function	(NULL)	One of GROW_SHIFT, GROW_FIXED, or GROW_FIXED_HORIZONTAL.
splash	Boolean	TRUE	If set to TRUE, briefly display logo window at startup.
suppress_table_warning	Boolean	FALSE	When TRUE, suppress the warning that occurs when the number of lines exceeds 1500.
uncomputed_height	Integer	150	Height allowed for uncomputed layout within a tab set.
uncomputed_width	Integer	450	Width allowed for uncomputed layout within a tab set.
user	String	"(NULL)"	Selects which report directories to use from the *system/user.dat* file. Pretend our user id is this user. Only works for the user who started the engine.
x_zoom	Float	1.25	Chart X axis zoom multiplier.
y_zoom	Float	1.25	Chart Y axis zoom multiplier.

2.4.5 Customize Options

Table 6 lists the engine and UI options that are used to customize RHYTHM SCP and that are common to both the engine and UI executables (*scp_engine help all*, *scp_ui help all*, and *scp_batch help all*). Some are only for one or the other of engine or UI but not both (and are marked as such).

Table 6: Customize Options - Engine and UI

Option	Type	Default	Definition
absolute_pathnames	Boolean	FALSE	If set to TRUE, prepend the current directory to relative pathnames. If set to TRUE, the $I2_REPORTS/ environment variable in worksheet messages is replaced by the actual pathname.
backup_prefix	String	""	When writing files, append this string to the beginning of the old file. Whenever a file is opened for write, and either this option or *backup_suffix* (or both) is a non-empty string, any existing file is renamed before the open smashes it.
backup_suffix	String	"~"	When writing files, append this string to the end of the old file. Whenever a file is opened for write, and either this option or *backup_prefix* (or both) is a non-empty string, any existing file is renamed before the open smashes it.
boolean_false	String	"0FfNn-"	The set of characters which convert to "FALSE." The first character is used for output.
boolean_format	Function	False, True	Default format for Boolean values.
boolean_true	String	"1TtYy+"	The set of characters which convert to "TRUE." The first character is used for output.
buffer_size	U-Integer	8192	ASCII file buffer size.
char_format	Function	%c	Default format for characters.
classpath	String	"$I2_HOME/ i2.jar:$I2_HOME/ scp.jar:/opt/ jre1.1.5/lib/rt.jar"	Where to find Java classes $environment variables are substituted.
debug_prompt	String	"Break {0} > "	Prompt string for the OIL debugger. If the string contains '{0}', it will be replaced with the breakpoint name.
default_format	String	"default"	The default format to use for all controls.
default_style	String	"Default"	Style to use for attributes which nothing else specifies.

Table 6: Customize Options - Engine and UI

Option	Type	Default	Definition
deleted_error_display	String	"<DEL>"	What to display in a worksheet for the DELETED value.
do_file_debug	Boolean	FALSE	If TRUE, do_file reads at runtime instead of compile time.
doc_changed_p	Boolean	FALSE	Must be set to TRUE to document changed strings.
file_type	String	"(NULL)"	Default datafile extension (e.g. <dir>/<file>.<file_type>). Example: dat
font_increment (UI only)	Integer	0	Integer by which to increase / decrease all font sizes.
galaxy (UI only)	Function		Arbitrary Galaxy options.
gc_threshold	U-Integer	10	Garbage collection threshold in megabytes. When exceeded, unused report memory is cleaned up.
hex16_format	Function	0x%04x	Default format for 2 byte hexadecimal numbers.
hex32_format	Function	0x%08x	Default format for 4 byte hexadecimal numbers.
hex8_format	Function	0x%02x	Default format for 1 byte hexadecimal numbers.
hex_format	Function	0x%x	Default format for hexadecimal numbers.
int_format	Function	%d	Default format for integers.
max_undo_changes	U-Integer	10000	Maximum number of plan changes to undo. Higher limits cause more memory usage.
new_user	String	"New_User"	All new users inherit from this user. The default New_User gets its directories from the [unspecified] user. When *scp_ui* runs, and there is no entry in the user database for that user, the path in *new_user* is copied. *new_user* can be useful when all users are to be started with a set of default reports. See separate section titled User for more details.
nonexistent_error_display	String	""	What to display in a worksheet for the NONEXIST-ENT value.
ptr_format	Function	0x%08x	Default format for pointers.
recurse_depth	Integer	1000	Maximum recursion depth for the 'recurse' function.
recurse_items	Integer	10000	Maximum number of items computed in a recursive function before it truncates.

Table 6: Customize Options - Engine and UI

Option	Type	Default	Definition
reference_error_display	String	“<REF>”	What to display in a worksheet when using a cell value which has an error.
report_parse_errors	Boolean	TRUE	Must set to FALSE to turn off parse error reporting.
seed	Function	TRUE	seed for rand() functions. If set to TRUE, this option causes the random number generator to produce a consistent pattern each time it is run. The random number generator is used in making arbitrary choices.
specfile_type	String	“spc”	File type (extension) for specfiles.
strict_conversion	Boolean	FALSE	If set to FALSE, string->number conversions can have garbage at the end of the string.
tab_width (UI only)	Integer	8	Default tab width, in characters.
timezone	Function	(NULL)	Set the Local Timezone. 0 is UTC (Greenwich time), 6 is CDT (Central Daylight). This option will more often be applicable to NT systems than UNIX systems because NT systems are sometimes not configured for the correct timezone. The default comes from the environment variable $I2_TIMEZONE. If this variable is not defined, then the default comes from the system.
uncomputed_error_display	String	“<>”	What to display in a worksheet for cells which are not computed.
update_interval	Float	3	Minimum of seconds before windows are automatically updated.
value_error_display	String	“<VAL>”	What to display in a worksheet for internals errors (overflow, etc.).
while_max	Integer	4000000	Maximum number of while loops before infinite loop error.
worksheet_error_display	String	“<ERR>”	What to display in a worksheet for worksheet formula errors.

2.4.6 Connecting Multiple UIs to an Engine

To authorize a client to connect to your machine, you can give him/her permission using the *system/user.dat* file. For example, when your loginid on UNIX (where you run the engine) is john whereas your loginid on NT (where you run the client) is "John Doe" or "john doe". You would modify the *system/user.dat* and copy the New_User line as follows:

New_User;	*New User;*	*[unspecified];*	*;*	*[unspecified]*
John Doe;	*New User;*	*[unspecified];*	*;*	*[unspecified]*
john doe;	*New User;*	*[unspecified];*	*;*	*[unspecified]*

2.4.7 Security

All users of the UI must be registered in the *system/user.dat* file except the *super-user* (the user who started the engine). The *super-user* can run as anyone by using the *user* command line option.

If a user's name does not appear in *system/user.dat* and a default New_User is not defined, then he is not given access to the engine. This is a security feature, as illustrated by the following example:

Suppose a company is running two engines. Some users need to be restricted from accessing one of those two engines. This restriction is accomplished by forcing all users to be registered in *user.dat*. The only exception is the user who starts the engine. That user must have write permission to *system/user.dat* (or the save/restore file) in order to add new users.

2.4.8 Server Timeout

To specify a time-out period for the server in case it should hang, you should use the *server_timeout* option. For example, in the following batch command, there is nothing to save if the engine is not running. The command hangs waiting for the engine, but the batch command times out after 120 seconds because the original command was run with the *server_timeout* option (default is 120 seconds):

Original command:

scp_ui server_timeout

Batch command:

scp_ui batch 'save("scp_" & now & ".i2")'

Waiting for server on port 27111

....

....

Error: Couldn't connect to server on port 27111 after 120 sec onds

2.5 Debugging in OIL

Several helpful options are available for debugging purposes in OIL. These options debug faulty OIL code while RHYTHM SCP is running. **Note:** Appendix C has more information on debugging in OIL. The options include the following:

- Selecting a cell in layout and pressing <Shift><F12> (invoking an OIL listener). This suspends the operation of the engine, producing an i2> prompt to handle a limited set of debug commands, or evaluation of variables in the layout. Commands available include the following:

  • print_variables
  • print_report_variables
  • inspect(model_instance)

- Setting flags in *scp_engine.opt*, such as:

  • command_execute: true;
  • action: true;

  These flags cause additional information to be printed when actions are taken in the UI. A few other helpful debug functions include the following:

- trace_usage (...) - this function is similar to the do function. It executes all of its parameters and returns the result from the last expression. In addition, it prints usage statistics about what was executed.
- trace_time (...) - this function is similar to trace_usage except that it only prints the time taken to evaluate its arguments.
- trace (String, ...) - this first parameter is a string which contains a list of trace flags to turn on or off. The rest of the parameters are expressions to evaluate the trace flags set. The trace flags are reset when the other parameters have finished executing.
- trace_echo (String, ...) - this function is used to print "Starting <string>" before evaluating the parameters or print "Finished <string>" after evaluating the parameters.
- report_text (String report_name, ...) - this function is used to print a report to stdout.
- layout_text (String layout_name, ...) - this function is used to print a layout to stdout.

- inspect (model_instance) - this function, given a model instance, prints all of its fields and their values.
- echo - this function displays the result of any expression as long as it evaluates to a string or a symbol. Debugging with `echo` is good for layered debugging, tracking flows, and time stamping (use with the `now` function).

  Examples:

  `buffer_plans.for_each(#.echo)` returns the key field

  `echo(true)` echoes every statement and its results
- do_echo - this function displays the results of each of the expressions within a `do`. Debugging with `do_echo` allows you to gather timing and memory metrics for `do` statements.

Section 3

Building a Report

The first item of business is learning to build a report. Reports, as you have learned, are mechanisms for calculating and organizing planning data. They are created and managed in OIL and consist of two elements:

- Layouts
- Worksheets

This section provides you with the information required to create a layout file (*.lyt*), a worksheet file (*.wrk*), and a report file (*.rpt*) for displaying the planning data.

3.1 Worksheets

Worksheets are the computational platform for RHYTHM SCP. They are analogous to spreadsheets in that they are two-dimensional tables of cells. The cells within a worksheet contain values computed by OIL expressions. *Worksheets* primarily deal with data, however, they can also accept parameters. *Worksheets* differ from spreadsheets in that they do not specify how data are input, printed, or displayed. The RHYTHM SCP server holds the actual models and the *worksheets* are interposed between the server and the outside world (e.g. humans, data files). Each *reports* subdirectory contains ASCII files that are parsed and compiled as needed into RHYTHM SCP reports for that client/server combination. *Worksheets* are defined (having the extension *.wrk* or *.fws* for functional worksheets) in those files.

3.1.1 Worksheet Types

Three types of *worksheets* define data for use in OIL:

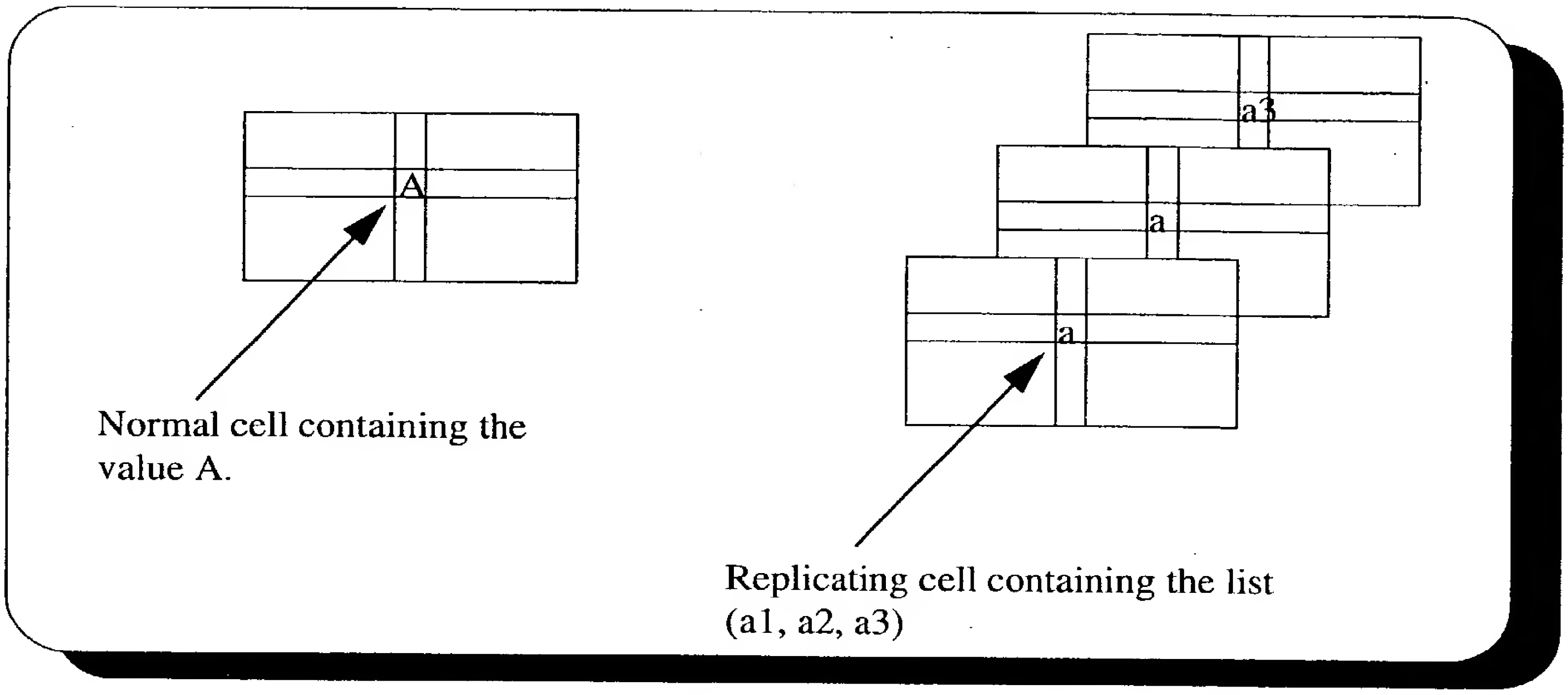
- Normal
- Replicating
- Functional

A *normal worksheet* has cells which hold non-list values. It holds data such as *strings*, *supply chains*, *#*, etc. *Normal worksheets* are the default worksheet type.

A *replicating worksheet* has cells which contain lists of values. Each cell within a *replicating worksheet* behaves like a normal cell that has been replicated, with each "copy" holding an element in the list. FIGURE 14 illustrates the difference between normal and *replicating worksheets.*

A *functional worksheet* behaves similarly to a replicating worksheet in that its cells can contain lists of values. However, the cells of a functional worksheet also execute sequentially.

FIGURE 14 Normal vs. Replicating Worksheets



3.1.2 Worksheet Cells

Worksheet *cells* are the basic component of a worksheet. They are similar to traditional spreadsheet cells and are accessed via alphanumeric cell IDs (e.g. A1, J26). Worksheet *cells* can hold any RHYTHM SCP data as OIL expressions.

In a normal worksheet, a cell can contain only a single value. In a replicating worksheet, a cell contains a list (even if it looks like a single value). A functional worksheet cell can contain either a single value or a list.

3.1.2.1 Cells and Get/Set Expression

Cells have expressions called *get* and *set* expressions. The *get* expression places a value into the *cell*. This value comes from either the engine or the outside world. Each *cell* **must** have a *get* expression even if it only returns *nonexistent*. The *set* expression writes a value from the *cell* to the engine. Explicit *set* expressions are not required. If a *set* expression is not present, RHYTHM SCP tries to deduce it from the *get* expression. For example, if the outermost function is a settable field, RHYTHM SCP uses the *set_* function to set it. If RHYTHM SCP cannot deduce the *set* expression, the *cell* is uneditable.

get expressions are evaluated whenever:

- importing data
- interactive editing of planning data
- exporting data
- reading "read-only" portions of the user interface

set expressions are evaluated whenever:

- importing data
- interactively editing planning data

Note: Since the *set* expression can be any valid OIL expression, it is possible to invoke all sorts of server and client behavior based on user inputs.

3.1.2.2 Get/Set Expression Syntax

The proper syntax for *get* and *set* expressions requires a *cell id* and a *get* expression. The *set* expression is optional (as discussed previously). The syntax is shown below. **Note:** Interactive entries and imports are denoted in the *set* expression by #. Also, expressions within the < > are optional.

```
[ cell = get_expression <, set_expression >; ]
```

For example,

```
[ B3 = site.description; ]// B3 holds the
          value of site.description
```

Here, the *get* expression is `site.description` and no *set* expression is provided. However, OIL will deduce a *set* expression as follows:

```
[ B3 = site.description, site.set_description (#); ]
```

so that if you modify cell B3, that data will be used to set the description field of site.

As another example, consider holding the percentage of the committed forecast in an editable cell:

```
[ A1 = percentage(forecast_entry.committed /
     forecast_entry.forecasted); ]
```

Note that in the expression above, the cell is not editable. You must explicitly define a *set* expression so that a user can enter a percentage and have it update the committed field. For example,

```
[ A1 = percentage(forecast_entry.committed
     / forecast_entry.forecasted),
        //set expression forecast_entry
                   .set_committed(forecast_entry
                   .forecasted * percentage(#)); ]
```

3.1.3 Dependent vs. Independent Cells

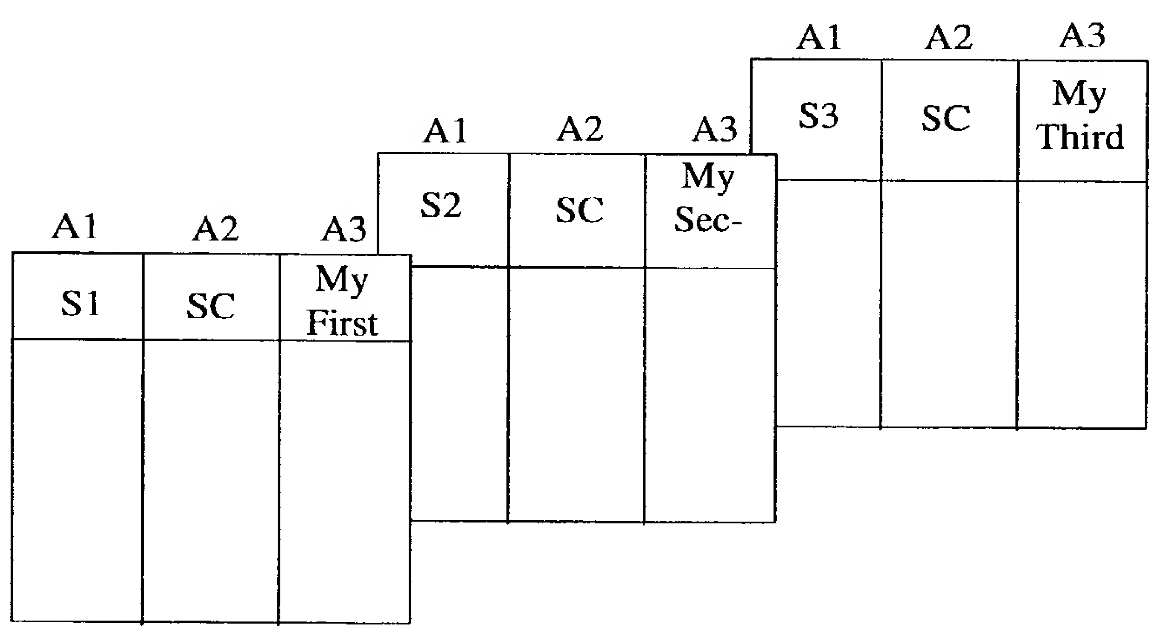
A cell is dependent if its computed value depends on the value of another cell in the worksheet. For example,

```
[ A1 supply_chain = supply_chains.first; ]
[ A2 name = supply_chain.name; ]
[ A3 desc = supply_chain.description; ]
```

Cells *A2* and *A3* are dependent on the value of cell *A1* and are hence *dependent cells*. Cell *A1* is an *independent cell.*

In replicating worksheets, cell dependencies determine what layer the values reside on. If you have supply_chains named S1, S2, and S3, and a replicating worksheet such as:

```
[ A1 supply_chain = supply_chains.first; ]
[ A2 name = supply_chain.name; ]
[ A3 desc = supply_chain.description; ]
```



3.1.4 Worksheet Syntax

Worksheets require a particular syntax. This syntax is shown below:

```
class name (<Type var1, Type var2... >) {
    <declarations;>
    .....
    [ <cell_id> <cell_name> = expression <;> ];
    .....
}
```

where *class* identifies the type of the worksheet (normal, replicating, or function procedure) and *name* identifies the name of the worksheet (identical to the *.wrk* file name). The *name* of the worksheet can be omitted only if the layout specifies the worksheet to use (e.g. *import_text_file* layouts).

Cells may be assigned local cell names that act as aliases to the cell ids. If omitted, cell ids are automatically generated as the next cell in the current default column. **Note:** The cell id syntax from import files is NOT used outside of import files. For example, write **A1** not **A1:**. Also, do NOT use names such as A1, C12, K7, etc., to name anything but cells.

FIGURE 15 displays the contents of *site_context.wrk* which illustrates the proper syntax for a worksheet.

FIGURE 15 Worksheet Syntax Example: site_context.wrk

```
normal site_context (Site site)
{
[ A1 = "The Site:"; ]
[ B1 = site; ]
[ C1 = "of Supply Chain:"; ]
[ D1 = B1.owner; ]
[ E1 = "of engine:"; ]
[ F1 = engine_name; ]
}
```

Note: Unless otherwise specified, filenames cited in examples are in the *.../reports/scp/* directory.

3.1.5 Worksheet Parameters

Worksheet parameters have the following properties:

- Allow the worksheet to act as a template; the worksheet works for any values of the specified types.
- Are optional (e.g. if the worksheet is storing static information like a list of menu options).
- Are treated as local variables to the worksheet.
- Are the means of transferring information between worksheets.

FIGURE 16 shows an example of worksheet parameters, where the normal worksheet named, "Foo", has two parameters, *x* and *n*.

FIGURE 16 Parameters Example

```
normal foo (List[Site] x,
Number n)
{
[ A1 = x.element(n) ;]
}
```

In the example above, *x* is a list of *Site* models, *n* is a *number*, and *A1* will contain the *n*th site in *x*.

3.1.6 Functional Worksheets

Functional worksheets behave as replicating worksheets: their cells can hold replicated lists of values, and they can accept parameters. But unlike other worksheets, functional worksheets are designed to be directly invoked as functions from within any OIL expression. This is done using the OIL *call* function, which returns a result that is calculated by the invoked functional worksheet.

Functional worksheets are the preferable method of creating OIL scripts (i.e. collections of OIL computations that can be invoked from batch clients). They can also be easily invoked from within the OIL listener by using the *call* function. In this way, complex computations can be prototyped in a functional worksheet and quickly tested from the listener.

The following example shows the contents of a file, *first_op.fws*, that defines a functional worksheet:

```
function first_op(String sc_str, String site_str)
{
[A1 SC = supply_chains.find(sc_str);];
[B1 MYSITE = SC.sites.find(site_str);];
[C1 return = MYSITE.operations.first;];
}
```

All functional worksheets impose a sequential ordering on evaluating cell *get* expressions. In this example, the worksheet accepts two string parameters, and then uses them to calculate the contents of cells A1, B1, and C1 in sequential order.

3.1.6.1 call Function

The example functional worksheet above could be placed in a *custom_reports* directory and invoked from within an OIL expression as follows:

```
call("$I2_REPORTS/custom_reports/first_op",
                "My Supply Chain", "My Site")
```

The *call* function's first parameter names the *.fws* file that contains the function worksheet to be invoked, and the remaining parameters are passed to the worksheet itself. The result of the call is the Operation model calculated in the return cell (cell C1 in this case).

3.2 Layouts

A *layout* is a collection of *controls*. The type of the *layout* determines the type of *controls* allowed (e.g. *bar_chart* understands bar controls but not button controls). The *control* determines the properties of the associated worksheet cell in the parent *layout*. Common *controls* include *button*, *checkbox*, and *general*.

A *layout* identifies which data is going into a report and how it will be displayed. The displayed data is contained within the worksheet file referenced by the *layout*. Each *layout* refers to exactly one worksheet. A worksheet can be shared by several *layouts* (for data and computations). For example, the same data (Resource Utilization) can be displayed as a table or a bar chart. *Layouts* are parameterized via the underlying worksheet. Thus, *layout* parameters are really just the worksheet parameters.

Layouts can be nested inside each other. A layout that contains other layouts is called a *parent* layout. The layouts that are nested within a parent are *child* layouts.

3.2.1 Layout Cells

There is a one-to-one correspondence between the *layout cells* and the cells of the worksheet that the *layout* uses. *Layout cell* definitions are used to specify the display and *layout* properties of the cells. *Layout* definitions include the following aspects:

- Controls - determines the display of the cells within reports
- Actions and Bindings - determines utilities attached to cells
- Styles and Formats - affects how the values in the cells are displayed (colors, number of digits, ...)
- Properties - defines various cell aspects (e.g. the protected property defines cells as uneditable). Properties include width, height, etc.

3.2.2 Specifying Layout Syntax

Layouts do not define their own parameters. Since a *layout* refers to, at most, one worksheet, it passes arguments to that worksheet from the report that contains the *layout* (see 3.1.6.3 for an example). *Layouts* require a particular syntax. This syntax is shown below:

```
layout_type name
{ worksheet: ws_name;
  <property: value; >
  <action: action_name = expression;>
  <bind: keystroke = action;>
  [ cell_id = <action: event = expression;>
     <control: control_name (arguments);>
     <bind: event = action;>
     <style: style_spec;>
     <format: format_string;>
     <protected: logical;>
     <...> ]
......}
```

where *layout_type* identifies the type of the *layout*, *name* identifies the name of the *layout* (identical to the name of the *.lyt* file), and *worksheet* declares the name of the associated worksheet, *ws_name*. **Note:** Often the *layout* (*name*) and *worksheet* (*ws_name*) names are the same, but this format is not required when creating *layout* and *worksheet* files. Also note that there can be multiple *property*, *action*, and *bind* statements. These statements do not necessarily have to be in that order.

FIGURE 17 displays the contents of *active_strategy_context.*lyt which illustrates the proper syntax for a layout. *active_strategy_context.*wrk is also shown to illustrate the relationship between layout and worksheet files.

FIGURE 17 Layout Syntax Example: active_strategy_context.lyt

```
spreadsheet active_strategy_context
{ worksheet: active_strategy_context;
[ A1 = style: Context_Label; ]
....}

normal active_strategy_context (Active_Strategy active_strategy)
{
[ A1 = "The Active Strategy:"; ]
....}
```

Contents of active_strategy_context.lyt

Contents of active_strategy_context.wrk

The layout is of type spreadsheet. This means that the worksheet data will display as a two-dimensional table, with cell IDs determining data placement. In the example above, the string *A1* displays using the *context_label* style and is located at the upper left-hand corner of the layout.

FIGURE 18 displays the contents of *seller_context.lyt* as another example of a *spreadsheet* layout's syntax. This displays a horizontal row of data, A1 through D1.

FIGURE 18 seller_context.lyt Example

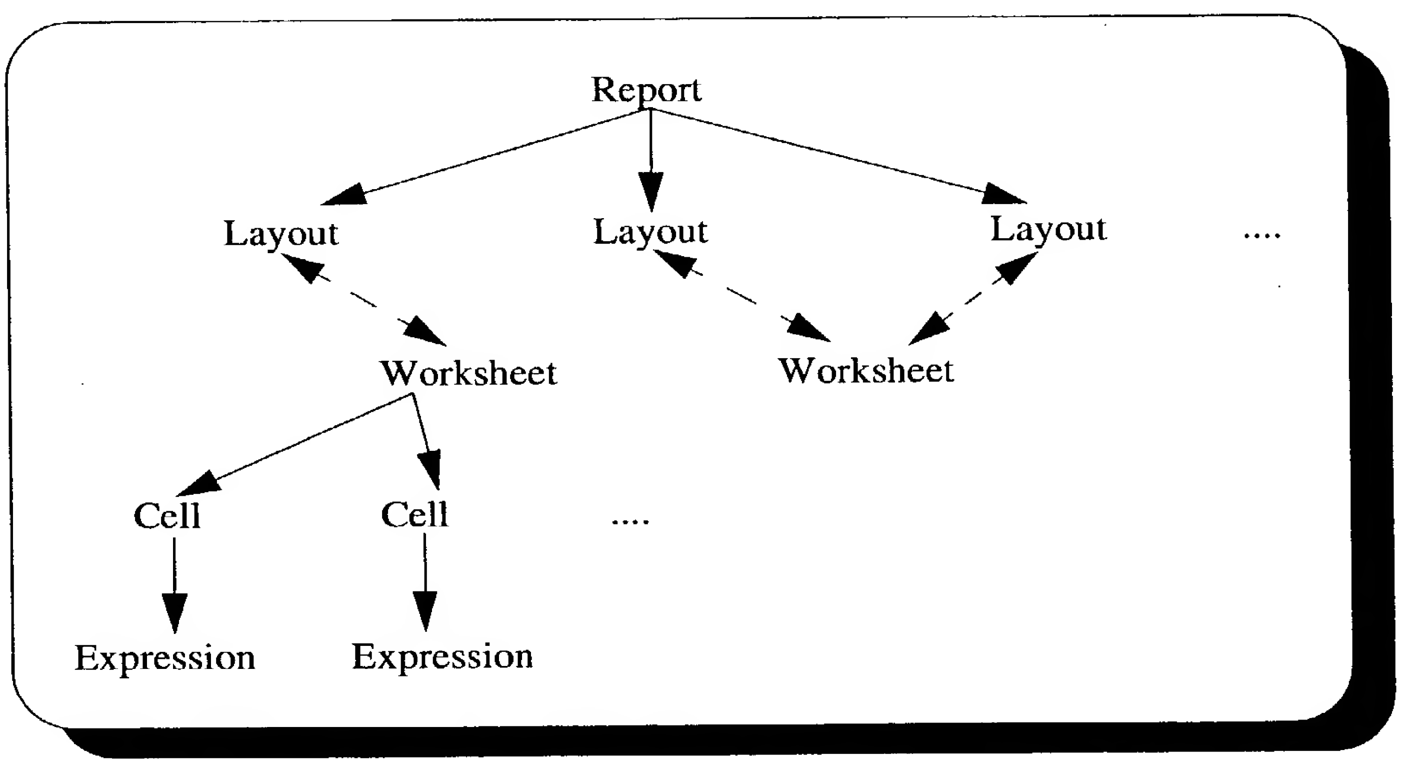
```
spreadsheet seller_context
{
worksheet: seller_context;
[ A1 = style: Row_Title; ]
[ B1 = action: report = dispatch("model_edit"); ]
[ C1 = style: Row_Title; ]
[ D1 = action: report = dispatch("model_edit"); ]
}
```


3.3 Reports

Reports combine the worksheet and layout elements to define GUI Windows, Text Reports, and other client interfaces. *Reports* have parameters which allow the definition of the *reports* to be read and used by RHYTHM SCP without knowing ahead of time the specific model instances with which it will be working. The instance of the *report* created depends on the actual values that are passed as arguments to the *report*. *Reports* are maintained as ASCII files with the extension *.rpt*.

FIGURE 19 shows the relationship between reports, layouts, and worksheets.

FIGURE 19 Interaction between Reports, Layouts, and Worksheets



3.3.1 Embedding vs. Sharing OIL Elements

Alternate syntax can be used to embed worksheets within *.lyt* files and layouts within *.rpt* files.

3.3.2 Specifying Report Syntax

The syntax of *reports* is similar to that of import files. It should be noted that the layout declarations **must** be specified at the end of the file. This syntax is shown below:

```
name (parameters)
{
   property: value;
   variable variable_name = expression;
   compute compute_name = expression;
   action: action_name = expression;
   bind: keystroke = action;
   #include include_filename;
   layout: layout_name(parameters);
   [ layout properties ];
   ......
}
```

Note: There can be multiple *property*, *variable*, *compute*, and *action* statements. These statements do not necessarily have to be in that order.

FIGURE 20 shows the contents of *all_probs.rpt* which illustrates the proper syntax for reports.

FIGURE 20 Reports Syntax Example: all_probs.rpt

```
all_probs (Plan plan)
{
title: "All Problems";
layout: scp_menubar ();
layout: scp_file_menu () [visible: FALSE; ];
layout: supply_chain_context(plan.owner);
layout: problem_list_late(plan.the_problems);
}
```

In the example above, RHYTHM SCP will look for the corresponding *.lyt* files in the $I2_Reports directory; if not found, the software will search other directories according to the sequence specified in *user.dat*. This same search approach is always used for OIL files (*.rpt*, etc.).

The first line of the example specifies an expression that sets the title for this report. The remaining lines specify the order of layouts, one above the other, from top to bottom.

The *scp_file_menu* layout will not be visible when the *all_probs* report is initially displayed. Therefore, you will see three horizontal "strips": *scp_menubar*, followed by *supply_chain_context* underneath it, followed by *problem_list_late* at the bottom.

3.3.3 Reports Parameters

Just like worksheets,

- *Reports* take any value type as a parameter.
- There is no limit on the number of parameters *reports* can take.
- The parameters are used as local variables in that *report.*

For example:

```
plan_editor (Plan the_plan) {
.....
layout: my_layout (the_plan);
.....
}
```

In the example, *the_plan* is used as a local variable.

Names of variables that are used to pass values to worksheets need not match the names of the worksheet parameters. For example:

```
my_report (Number x) {layout: my_layout (x); }
-------------------------------------------------------------
spreadsheet my_layout { worksheet: my_worksheet; }
-------------------------------------------------------------
normal my_worksheet (Number y) { [ A1 = y; ] }
   //y will get the value of x
```

In the example, the parameter *x* of *my_report* will be passed to *y* in *my_worksheet* via the invocation of *my_layout.*

3.3.4 Invoking Reports

Reports are displayed by invoking the *display_report* function *(display_report ("report_name", parameters);)*. Example uses of this function are shown below.

- with actions defined in cell controls

```
[ E1 = action:report = display_report("plan_editor",
                                       supply_chains.first
                                       .plan.first); ]
```

- on UI startup

```
scp_ui \
   initialize 'display_report("plan_editor", \
                  supply_chains.find("Super Metals")
                  .plans.find("Actual Plan"))'
```

3.3.5 Reports of Importance

Two standard (i.e. i2 written) *reports* of special importance exist. These reports are listed below:

- *application.rpt* - found in the *basic reports* directory. It contains most of the default and standard actions, definitions, and bindings. All *reports* inherit the actions, bindings, and variables defined here. **Note:** There is no screen associated with *application.rpt*.
- *main.rpt* - the default main window definition (may be overridden using the *scp_ui initialize* option).

Note that *read_changed_oil_files* does not work with these two reports only. The only way to see changes to these two reports is to restart the engine.

3.3.6 Scoping Rules

Reports can only access the following data:

- input parameters
- *variable* and *compute* declared within the report
- parameters passed into the report

Worksheets can only access the following data:

- input parameters
- *variable* and *compute* declared within the worksheet
- parameters passed into the layout

Layouts can access the following data:

- all the data from their associated worksheet
- input parameters
- *variable* and *compute* declared within the worksheet

Report parameters are passed by reference. Therefore, each *report* gets its own copy of the input parameters. This limits the changes a *report* can make to be within its own scope.

Worksheet parameters, in contrast, are passed by reference. Therefore, if the same value is passed to more than one *worksheet*, then *worksheets* share data. This allows a change made in one *worksheet* to affect other *worksheets* in the report (i.e. the *worksheets* stay in sync).

You might refer to our parameter passing as "conditional reference". If the actual parameter is a variable name, the parameter is passed by reference; otherwise, it is passed by value. This is true for all calls that pass parameters.

In the example shown in section 3.3.3, if *x* changes its value within *my_report*, then *y* will change its value within *my_worksheet*.

3.4 Computed Properties

Most properties in worksheets, layouts, reports, import worksheets, and export worksheets can be expressions. For example, you can pass the filename(s) to be read from to an import worksheet as a parameter. OIL automatically tries to figure out which properties are expressions and which are constant, but sometimes it cannot tell. To force a property to be computed, use two ":"s instead of one. For example, instead of:

```
file : file;        // File will be "file"
```

use

```
file :: file;       // File will be the value of the 'file' variable
```

The following is an example if passing a filename as a parameter to an import worksheet:

```
     ----- File: supply_chain.imp -----

import_text_file supply_chain
{
worksheet (string file)
{
     model: "Supply_Chain";
     this = supply_chains;

     [A1: name = #;]
     [B1: description = #;]
}
file :: file;
import_record: A1 B1;
}
```


3.4.1 List of Computed Properties

The following properties can be specified as computed properties:

Property	Control
visible, compute_when_visible, count, disable, protected, growth_factor, default_rows, default_cols, point_width, point_height, description, remark, no_scroll, x_scroll, y_scroll, x_stretch, y_stretch, multi_select, tab_set, tab_title, tab_icon, layout_name, width, height, dynamic_cell_sizes	layout
x_synch, y_synch, x_range, y_range, hide_left_axis, hide_right_axis, hide_top_axis, hide_bottom_axis	univ_chart
y, x, cross, sort, sparse, total_down	axis_cross
always_recalc, title, icon, description, remark, fast_update, auto_update, hide_disabled_layouts, modal	report
title, icon	application
import_record, import, delimiter, file, fixed_width, sequence, optional, before_import, after_import	import_text_file
import_record, import, sequence	import_rhythmlink
delimiter, file, fixed_width, before_export, after_export, title_line_prefix, sequence	export_text_file
x_synch, x_range, y_synch, y_range, hide_bottom_axis	gantt_chart
x_synch, y_synch, y_range	bar_chart
foreground_color, background_color, format, border_color, pattern, pattern_color, h_align, v_align, invert, font_family, font_style, font_size, border_style, border_top, border_right, border_bottom, border_left, width, height, disable, hide, protected, modal	style
foreground_color, background_color, font_family, font_style, font_size, disable, hide, width, height, protected, left_inset, right_inset, top_inset, bottom_inset	item (**Note:** these dialog items only support a subset of style)
image_name, image_position	image_general
max_initial_level	outline_general
image_name, vertical, center, draw_label	button

Property	Control
image_name, action_name, tip	tool_button
dirty_image_name	update_tool_button
layout_name, exclusive	combo_popdown
layout_name, exclusive	computed_combo_popdown
minimum, maximum	slider
minimum, maximum	spinner
label, true_label, false_label	checkbox
label	radio_button
action_name, label, license_package	menu_item
layout_name, label	submenu
layout_name	use_layout
bar_label, bar_color, width, height, stack_set	bar
y, x_synch, y_synch, x_range, y_range, sort	line_chart
format	secondary_axis
format	time_buckets

3.5 Guidelines for Formatting Reports

The following report formatting guidelines enable the user to write reports which are efficient and easy to read.

3.5.1 Layout Declaration and Invocation

When defining a layout function within the *.wrk* file, place the first parameter on the same line as the layout name and all succeeding parameters directly below the first parameter, one parameter per line. For example,

```
replicating mpp_items_category_summaries (Plan plan,
                                          Site_Plan mpp_site_plan,
                                          Item mpp_item,
                                          Symbol category,
                                          List[Date_Range] bucket_list,
                                          Logical l_demand,
                                          Logical l_starting_oh,
                                          Logical l_planned,
                                          Logical l_change,
                                          Logical l_ending_oh)
```

The look of the layout when called from the *.rpt* files should be similar:

```
layout: mpp_items_category_summaries (plan,
                                      mpp_site_plan,
                                      mpp_item,
                                      item_category,
                                      bucket_list,
                                      l_demand,
                                      l_starting_oh,
                                      l_planned,
                                      l_change,
                                      l_ending_oh)
        [ tab_set:     tabs;
          tab_title:   Items of a Category;
          count:       nonexistent;
          tab_icon:    product;
          disable:     FALSE; ];
```

The advantages of following these formatting tips are as follows:

- makes determining the number of arguments of a list and mapping their type from the calling location to the declaration easy
- makes separating the parameters from their values quick and simple in the invocation since the tab information is aligned
- makes scanning through the *.rpt* files to see how many layouts exist exponentially easier since the word *layout* is the only one in the left-most column area

3.5.2 Appearance of compute Functions

computes can be difficult to format efficiently within *.rpt* files. The following format works efficiently for *computes*:

```
compute check_for_site_plan =
      if(and(current_site_plan == mpp_site_plan,
            current_site_plan.owner == mpp_site_plan.owner),
         current_site_plan,
         do(set_variable(mpp_item,
            mpp_site_plan.site.items.filter(#.name ==
                                            mpp_item.name).first ?
            mpp_site_plan.site.items.first),
           if(mpp_item == nonexistent,
            set_variable(item_category, nonexistent)),
           set_variable(current_site_plan, mpp_site_plan)));
```

In the above example, the first line of the actual computation appears below the name of the *compute* itself. This provides between ten to thirty extra columns to work with and maintains the *compute* clear in the left column for easy visual access.

On a more general level, *if*, *and*, *do*, etc., statements should contain only one argument per line and they should be aligned to the "(" of the statement. This is a good general rule to follow (even if you can fit more than one argument on a line) because when the file is edited, the beginning of each line following a *do* statement can be quickly scanned to determine the arguments. **Note:** Some arguments will take up more than one line, as seen above. This cannot be avoided.

An eighty column maximum should be adhered to whenever possible, falling back to one hundred only when there is no clean way to break a statement up into two lines. This is done because eighty columns is a much more common standard than one hundred columns.

A final guideline for this section is to remember to use whitespace. Whitespace, when used properly, improves the presentation of information.

3.5.3 General Layout of .rpt Files

The following format should be followed when creating *.rpt* files:

```
report declaration()
{

      remark:

         yaddayaddayadda
         .....

      ; //end of remark

      [variable and compute section]
```

The *variable* and *compute* section should be staggered as follows:

```
variable current_site_plan = sp;
compute check_site_plan = if(current_site_plan != sp,
                      ...
```

If the *variables* and *computes* are grouped according to function rather than structure, the groups are easier to find and change when updates are needed. For example,

```
// Initializations

variable plan = sp.owner;
variable mpp_site_plan = sp;
variable mpp_item = mpp_site_plan.site.items.first;

// Synchronizations
variable current_plan = plan;
compute check_plan =
        if(current_plan != plan,
            do(set_variable(mpp_site_plan, plan.site_plans.first),
               set_variable(mpp_item, mpp_site_plan.site.items.first),
               set_variable(current_plan, plan))
        current_plan);

variable current_site_plan = mpp_site_plan;
compute check_site_plan =
        if(mpp_site_plan != current_site_plan,
            do(set_variable(mpp_item, mpp_site_plan.site.items.first),
            set_variable(current_site_plan, mpp_site_plan)),
        current_site_plan);
etc....

[ action section ] (if any, some have it some don't)

[ layout section ]
  [ menu includes ]
  [ invisible layouts ]
  [ context layouts ]
  [ selector layouts ]
  [ tabsets ]
```

include files should be placed in the most appropriate section. For example,

buckets.act would go in the actions section
planning.mnu would go in the context and selection section
main_report.var would go in the variable section

3.5.4 Placement of Comments

Each section shown in the previous example should have a brief one-line comment marking the beginning of the section. The action section should be further subdivided, as shown below:

```
//popup menu actions
  action: ...

//button actions
  action: ...
```

A remark should always be included.

3.5.5 File Naming

.rpt files consist of the full name of the report, with underscores separating words (i.e. *master_production_plan.rpt*). *.lyt* and *.wrk* files append the initials of the report name to the layout and worksheet files. This causes the files to be listed together in the directory list and makes it easier to determine which files go where. For example,

date_editor.rpt

de_month_year.lyt
de_month_year.wrk

Since not all report initials are unique (as with *master_production_plan* and *master_purchase_plan*) pick an arbitrary but logical set of initials to differentiate them (i.e. *mpp* and *mpc*). The only exceptions to this should be files that are used in more than one report, like *plan_aux_context*.

3.5.6 .wrk and .lyt Files Layout

The following guidelines should be followed for *.wrk* and *.lyt* files:

- Within *.wrk* and *.lyt* files, provide cells with names in addition to their location and refer to them by name in the *.lyt* files. For example:

in the *.wrk* file:

```
[ A1 bob = .... ]
[ B1 cooper = .... ]
[ C1 truman = .... ]
[ D1 leland = .... ]
```

in the *.lyt* file:

```
Y: bob cooper truman leland
```

Comprehending what is actually displayed is much easier this way.

- Place only one parameter/flag per line in the cell definitions:

```
[ E6 = width:  10;
       height: 10; ]
```

- Multi-statement actions should line up similar to the *compute* guidelines:

```
[ E3 =  action: select =
                do(set_variable(chosen_filter, "AUP"),
                set_variable(last_filter, chosen_filter),
                set_variable(chosen_rp, chosen_filter)); ]
```

- As a general rule, separate column cell declarations by at least one line:

```
[ C2 = action: select =
               do(set_variable(chosen_rp, "ARP"),
               set_variable(chosen_filter, "")); ]
[ C3 = action: select =
               do(set_variable(chosen_filter, last_filter),
               set_variable(chosen_rp, chosen_filter)); ]
[ D3 = control: radio_button(""); ]
[ D4 = control: radio_button(""); ]
[ D5 = control: radio_button(""); ]

[ E3 = action: select = do(set_variable(chosen_filter, "AUP"),
                           set_variable(last_filter,
                             chosen_filter),
                           set_variable(chosen_rp,
                             chosen_filter)); ]
```

- Some additions for OIL files in general:
  • Try not to make your lines more than 80 characters. Lines that wrap are hard to read and debug. Starting continued lines with a period enables a user to determine (at a glance) that they are continued from above.

  • This 80 character limit applies to comments as well. If the comment is going to be too long, break it and start another comment line.

  • Take into consideration that not all users looking at files have the same amount of experience. Comments should tell why as well as how.

  Some examples:

  // This expression will find and delete all the operation plans that were
  // generated to fulfill the requests that are going to be cancelled.

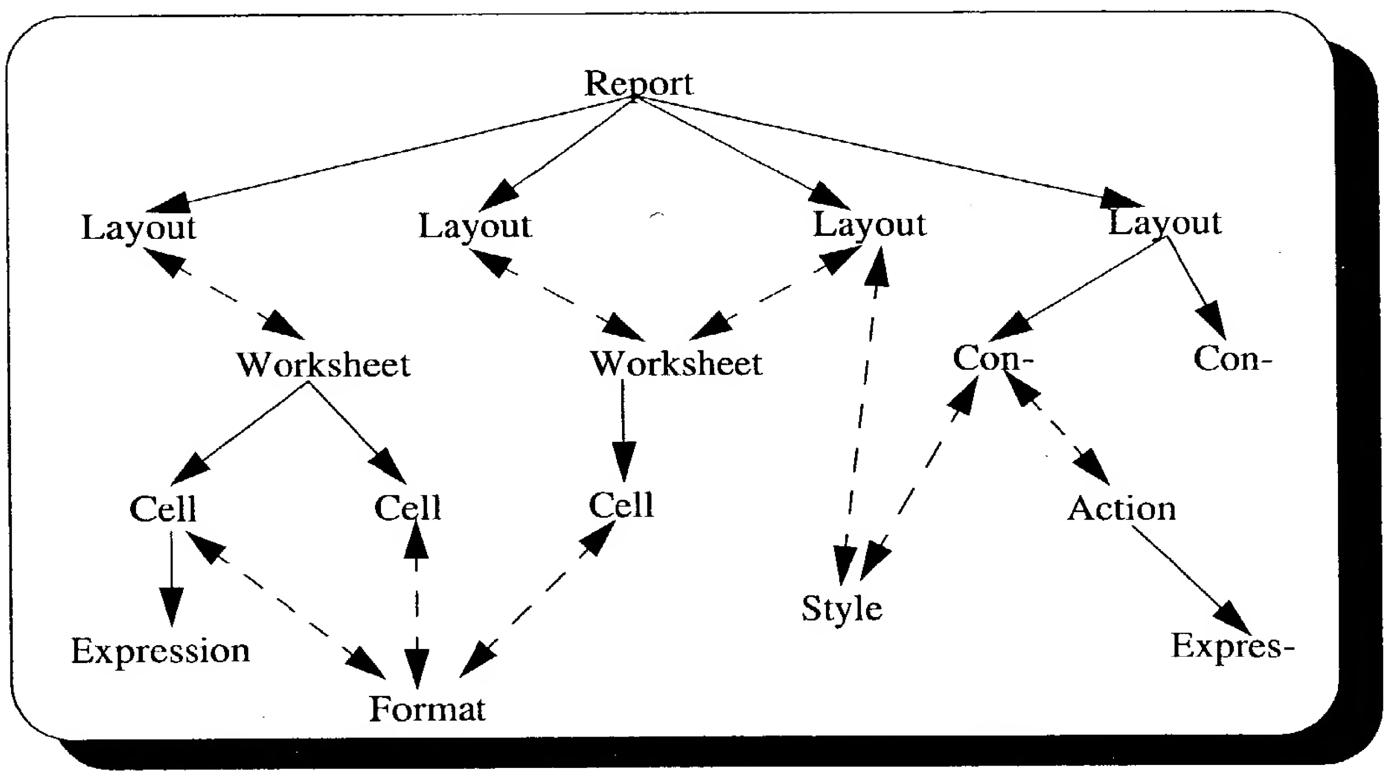
```
requests.for_each(#.delivery_requests)
         .for_each(#.item_requests)
         .for_each(list(#.delivery_plan
         .top_operation_plan,
         #.receiving_plan.top_operation_plan))
         .for_each(#.delete);
```


Section 4

Applying Controls, Formats, and Styles

You are now going to learn how to apply controls and formats to the basic reports you learned to create in section 3. (It is important to note that when using controls and formats the syntax **must** be exact. The parser will "stumble" over things such as a space between a declaration and its colon.) FIGURE 21 illustrates the relationship between reports and controls, formats, and styles.

FIGURE 21 Components of Reports



4.1 Controls

Controls are the individual display elements of a layout. The type of layout determines the controls available to comprise that layout. The control for a cell is responsible for the cell's rendering, whereas the cell is responsible for the data to be rendered.

Controls are broadly characterized as Chart, Table, or Text controls. (See the *RHYTHM SCP Model Reference Manual* for more information.)

4.1.1 Available Controls

The following is a list of all available controls:

bar	list_bar
button	map_connect
checkbox	map_node
combo	menu_item
combo_popdown	menu_radio_item
filler_bar	outline_general
gantt_axis	percentage_bar
gantt_bar	radio_button
general	separator
indented_general	slider
label	spinner
layout	submenu
line	time_buckets
line_rate	tool_button
line_step	

4.1.2 Common Controls

There are several controls that are commonly found or used within reports. Table 7 below lists these controls with a brief description:

Table 7: Common Controls and Their Descriptions

Control	Description

button	• Displays the cell as a *button* that can be depressed to invoke an action.
	• When given no arguments, this control displays the associated cell value as the button label.
	• When given a string and two Logicals, the cell does one of the following:
	  • displays the associated cell value as the button label
	  • displays the icon named by the string parameter
	  • if the first Logical is true, then the label appears below the icon; otherwise it appears to the right of the icon
	  • if the second Logical is true, then the icon and label appear centered inside the button
	• When this control is given a string and three Logicals, the behavior is the same as above except that when the third Logical is false, the button label does not appear at all.
	• The *button* is inactive (greyed-out) when there are no actions defined for it.

checkbox	• Displays as a square button that is toggled based on the associated cell value which must be of type Logical. • The *checkbox* is intended for use as "choose any of many". However, *check-box* by itself does not implement "choose any of many" behavior. (See *site_plan_editor.lyt* for an example of how such behavior is written in OIL.) • When the *checkbox* is invoked with no arguments, the checkbox titles are determined by the format for the type Logical. • When the *checkbox* is invoked with one String argument, the string is used as the checkbox label. • When the *checkbox* is invoked with two String arguments, the string are used as the on and off labels.

Table 7: Common Controls and Their Descriptions

Control	Description
combo_popdown	• Displays a list of values that the cell may be assigned in a *combo_popdown* list. • When invoked with no arguments, the list of values is determined by the value represented in the associated cell. For example, `[ A6 = my_site.role; ] // shows LINK, SUPPLIER, CUSTOMER` in the *combo_popdown* list. • When the *combo_popdown* is invoked with a single String argument, the list of values is determined by the layout named in the string. The layout is usually an axis cross with a single column. (See *resource_plan_choose_filters.lyt* for an example of how this behavior is written in OIL.)
general	• Displays the cell as an ordinary cell with the specified format (e.g. `general ("my_format")`). • The *general* control is the default control for spreadsheet and axis cross layouts. (See *resource_utilization.lyt* for an example.)
indented_general	• Displays the cell as an indented list. (See *group_forecasts.lyt* for an example.)
label	• Displays a String and does not cause any behavior on GUI events.
layout	• Displays layouts within layouts by specifying them as controls. • *Layout* accepts a parameter that is the name of a layout. • Allows layout arrangements other than the default top-to-bottom arrangement. (See *main_body.lyt* for an example.) • *Layout* allows for the inclusion of multiple layouts within a tab. (See *plan_problems_tab.lyt* for an example.)
outline_general	• Displays the cell as an indented list with outline controls (e.g. Main Explorer navigator).
radio_button	• Displays as a circular button that is toggled based on the associated cell value, which must be of type Logical. • Accepts a String parameter to be its label. If the parameter is the empty string "", then no label is shown. • *Radio_button* is intended for use as "choose exactly one of many". However, radio_button by itself does not implement "choose exactly one of many" behavior. (See *.../reports/basic/exit_dialog.lyt* for an example of how this behavior is written in OIL.)

Table 7: Common Controls and Their Descriptions

Control	Description
tool_button	• Displays as a button within a toolbar. • Can only be used in toolbar layouts. • Its three parameters identify the image, the action, and the tooltip string. (See *scp_plan_toolbar.lyt* for an example.)

4.1.2.1 Specifying Control Syntax

Controls require a particular syntax. This syntax is shown below:

control: control_name (control_parameters);

For example:

[A1 = control: slider(0, 1);];

There are exceptions to this syntax. They are as follows:

[**bar**(cell_id); <property>, <...>;];

[axis = **gantt_axis**(cell_id);];

[axis = **time_buckets**(cell_id);];

These are the only three controls that use different syntax.

4.1.3 Chart Controls

Chart controls are used in bar_chart, gantt_chart, and map_chart layouts. They can only be used within these types of layouts. The different chart controls are as follows:

- bar
- gantt_axis
- gantt_bar
- line
- line_rate
- list_bar
- map_connect
- map_node
- time_buckets

4.1.3.1 bar

The *bar* control is displayed as a bar in a bar_chart layout with a label and a value. The control expression for a *bar* control would be as follows:

```
[ bar(F1); style: Bar1; width: 30; ]
```

The entire set of bar controls might appear in a bar_chart layout named *revenue_cost_bar_chart*. The layout file *revenue_cost_bar.lyt* and its corresponding worksheet file are shown below.

```
//Layout
bar_chart revenue_cost_bar_chart

  worksheet: revenue_cost_bar_chart;

    [ axis: time_buckets(B2); ]
    [ bar(F1); style: Bar1; width: 30; ]
    [ bar(F2); style: Bar2; width: 40; ]

//Worksheet
replicating revenue_cost_bar_chart (Plan plan,
                                    List [Date_Range]
                                    bucket_list)

    [ head = "Revenue-Cost Bar Chart"; ];

    [ B1 dates = bucket_list; ];
    [ B2 dates = dates. start; ];

    [ F1 cost = plan.site_plans.filter (#.site
                        .managed)
                        .for_each(#.supply_request)
                        .filter(within
                        (#.accepted, dates))
                        .for_each(#.delivery_requests)
                        .for_each(#.delivery_promise)
                        .for_each(#delivery_price).sum; ];
    [ F1.title = "Cost"; ];
    [ F2 revenue = plan.site_plans.filter (#.site
                        .managed).for_each
                        (#.supply_request)
                        .filter(within(#.accepted, dates))
                        .for_each(#.delivery_requests)
                        .for_each(#.delivery_promise)
                        .for_each(#delivery_price).sum; ];
    [ F2.title = "Revenue"; ];
```

4.1.3.1.1 axis Property

One cell in the bar_chart layout must be preceded by the *axis* property. This specifies the cell values used to label the bottom *X axis* of the chart. Also, the remaining cells (used as bars in the chart) must be dependent on the cell used as the *axis*.

4.1.3.1.2 bar_color Property

For each bar within a bar_chart layout, if the *bar_color* value is set, its string value is used to set the foreground color of the bar. This property can be used in conjunction with the *stack_set* property to set the colors of the individual values.

4.1.3.1.3 bar_label Property

The *bar_label* property allows the user to set the label for the bar. If this property is not set, the label is derived from the bar value.

4.1.3.1.4 stack_set Property

Individual values displayed with the bar control may also be stacked using the *stack_set* property. Using the example from above, *D1* and *E1* could be stacked as follows:

[bar (D1); style: Bar1; width: 30; **stack_set**: “stack1”]
[bar (E1); style: Bar2; width: 40; **stack_set**: “stack1”]

All cells given a *stack_set* property with the same string value are stacked together.

4.1.3.1.5 Bar Values

At present, the only control allowed to follow the *axis* property in a bar chart is the *time_buckets* control. For displaying bar values, two controls are available:

- bar (value_cell) - plots a single cell value in a single bar
- list_bar (value_cell, name_cell) - plots a cell containing a list of values as a stacked bar using *name_cell* for colorization and popup tips.

The *bar* control is colored using the *style* property for that cell. The *list_bar* control expects a cell containing a list of values and will display a single bar with all values stacked. The second parameter must be a list of equal length containing strings used for colorization. The string is not a color name but the name of the object being plotted. For example, if the control is displaying the time used at a resource for producing a variety of products, the first *list_bar* parameter would be a time quantity and the second para-meter the name of the product. The *bar_chart* will then ensure that each product named *Widget1* displayed in any chart will be drawn with a consistent color.

4.1.3.1.6 Associated Actions

The following default actions are invoked from the *bar* control:

- Select
- Menu

4.1.3.2 filler_bar

filler_bar is a bar_chart control that behaves similarly to the *bar* control except that it is invisible and cannot be selected. This control is used within a bar_chart layout using the *stack_set* property to display an empty or blank space within the bar. The control expression for *filler_bar* would be as follows:

```
[ filler_bar(on_time); property: stack_set = "uno"; ]
```

The *filler_bar* control might be used in a layout name *filler_bar_chart.lyt*:

```
//Layout
  bar_chart filler_bar_chart
{
worksheet: simple_resource_load;

  [ axis: time_buckets(time); ]
  [ bar(early); property: stack_set = "uno"; style: Early; ]
  [ filler_bar(on_time); property: stack_set = "uno"; ]
  [ bar(late); property: stack_set = "uno"; style: Late; ]
}

//Worksheet
  replicating filler_bar_chart (Plan plan,
                                List [Date_Range]
                                bucket_list)
```


4.1.3.3 gantt_axis

The *gantt_axis* is a gantt_chart control that specifies the list of objects used to label the Y (vertical) axis, and contains the data referenced by the *gantt_bar* controls.

At present, the only control allowed to follow the *axis* property in a gantt_chart is the *gantt_axis* control. The control expression for a *gantt_axis* control might be as follows:

```
[ axis: gantt_axis(buffer); ]
```

The *gantt_axis* control might appear in a gantt chart layout named *buffer_gantt_chart.lyt*. The layout file *buffer_gantt_chart.lyt* and its corresponding worksheet file are shown below.

```
//Layout:
//
gantt_chart buffer_gantt_chart
{
   worksheet: buffer_gantt_chart;

   [ axis: gantt_axis(buffer); ]
   [ gantt_bar(operation, start, end, flow_plan);
      action: select = set_variable(dr,flow_plan.dates); ]
}

//Worksheet
//
replicating buffer_gantt_chart(List[Buffer_Plan] bps,
                                Date_Range dr)
{    [ head = "Gantt Chart"; ];
     [ buffer_plan = bps; ];

   [ buffer = buffer_plan.buffer; ];
   [ buffer.title = "Buffers"; ];
   [ flow_plan = buffer_plan.flow_plans(dr); ];
   [ operation = flow_plan.owner.operation; ];
   [ start = flow_plan.dates.start; ];
   [ end = flow_plan.dates.end; ];
}
```

4.1.3.4 gantt_bar

The *gantt_bar* control is a gantt_chart control used to specify the data used in plotting the bars within a gantt chart. The expression of the *gantt_bar* is a replicating worksheet, and is just one load plan when referred to in an action expression. The control expression would be as follows:

```
[ resource = rp.resource(); control: gantt_bar("");
      width: 24; ]
```

The Gantt_Chart lays out the bars in the minimum number of rows. This increases the effectiveness of the Gantt_Chart for shared use resources (the user can easily see where he is overloaded). This is done by sorting by start date and then laying out each bar in the lowest row whose last end data is before that start date.

The entire set of *gantt_bar* controls might appear in a gantt_chart layout named *resource_gantt_chart.lyt*. The layout file *resource_gantt_chart.lyt* and its corresponding worksheet file are shown below:

```
 // Layout:
//
gantt_chart resource_gantt_chart
{
 worksheet: resource_gantt_chart;
 action: on_layout_hide = set_variable(bucket,
                             buckets.first);
 [ axis: gantt_axis(resource); ]
 [ load_plan = action: select = set_variable(bucket,
                             load_plan.dates);
 action: move_in = resource_plan.move(load_plan,
                             true, false, false);
 action: move_in_out = resource_plan.move(load_plan,
                             true, true, false);
 action: move_out = resource_plan.move(load_plan,
                             false, true, false);
 action: move_in_or_off = resource_plan.move(load_plan,
                             true, false, true);
 action: move_out_or_off = resource_plan
                             .move(load_plan, false, true,
                             true);
 action: move = resource_plan.move(load_plan, true,
                             true, true); ]
 [ gantt_bar(operation, start, end, load_plan);
 action: select = set_variable(bucket,load_plan.dates);
 action: move_in = resource_plan.move(load_plan, true,
                             false, false);
```



```
  action: move_in_out = resource_plan.move(load_plan,
                              true, true, false);
  action: move_out = resource_plan.move(load_plan, false,
                              true, false);
  action: move_in_or_off = resource_plan.move(load_plan,
                              true, false, true);
  action: move_out_or_off = resource_plan.move(load_plan,
                              false, true, true);
  action: move = resource_plan.move(load_plan, true, true,
                              true); ]
}

// Worksheet:
//
replicating resource_gantt_chart(List[Resource_Plan] rps,
                                  List[Date_Range] buckets,
                                  Date_Range bucket)
{
     [ head = "Gantt Chart"; ];
     [ resource_plan = rps; ];
     [ resource = resource_plan.resource; ];
     [ resource.title = "Resources"; ];
     [ load_plan = resource_plan.load_plans; ];
     [ operation = load_plan.owner.operation; ];
     [ start = load_plan.dates.start; ];
     [ end = load_plan.dates.end;];
}
```

One cell in the gantt_chart layout must be preceded by the special *axis* property. This indicates that the cell values should be used to label the left axis of the chart. Also, it is required that the remaining cells (used as bars in the chart) are dependent on the cell used as the *axis*.

4.1.3.4.1 Parameters

For displaying gantt bar values, only the *gantt_bar* control is valid. It accepts four positional arguments:

- model_value - the model on which the model menu should act
- start_date - a date used as the gantt bar start value
- end_date - a date used as the gantt bar end value
- color_value - the tag used to select a color

When actions are run, the action *focus* is the cell specified by the *action_obj* parameter of *gantt_bar*.

4.1.3.4.2 Associated Actions

The following default actions are invoked from the *gantt_bar* control:

- Select
- Menu

4.1.3.5 line

The *line* control is a *line_chart* control. To get up-down-horizontal lines in a graph, use the *line_rate* control. If you use *line* or *line_step*, the graph lines may be sloping. The control expression for this line control would be as follows:

```
[ C2 = style: Bar1; control: line(); ]
```

The *line* control might appear in a line_chart layout named *buffer_on_hand_line_chart*. The layout file *buffer_on_hand_line_chart.lyt* and its corresponding worksheet file are shown below. Note that it is possible to have multiple *line* controls on a single row.

```
//Layout
    line_chart buffer_on_hand_line_chart

      worksheet: buffer_on_hand_line_chart
        [ C2 = style: Bar1; control: line(); ]
              //action: select = do(set_variable
                selected_bucket, fp.dates)); ]
        [ C2.title = style: Row_Title; ]
        [ D2 = style: Bar2; control: line(); ]
        [ D2.title = style: Row_Title; ];

      y: c2, c3, d2, d3;

    //Worksheet
    replicating buffer_on_hand_line_chart(Buffer_Plan bp,
                                 Date_Range selected bucket)

        [ head = "On-Hand Line Chart"; ];
        [ C1 fp = bp.flow_plans(selected_bucket); ];
        [ C2 date = if(fp.produced, fp.dates.end,
                                 fp.dates.start); ];
        [ C2.title = "Planned On Hand"; ];
        [ C3 on_hand - bp.on_hand(fp); ];
        [ D2 min_date = list(selected_bucket.start,
                                 selected_bucket.end); ];
        [ D2.title = "Min On Hand"; ];
        [ D3 min_on_hand = do(min_date, bp.buffer
                                 .min_on_hand()); ];
```

4.1.3.5.1 Associated Actions

The following default actions are invoked from the *line* control:

- Select
- Menu

4.1.3.6 line_rate

line_rate is a line_chart control which consumes three values (that of the current cell, and those from the next two columns). The values are: *x*, *y*, and *rate* (i.e. slope). These values are consumed in the stated order. For each point, a line is drawn from the previous *x/y* point to the current *x* at the slope specified by the previous *rate* value. From there a vertical line is drawn to the current *y*.

Note: The quantity units for *line_rate* must be appropriate for the math required. For example, if *x* is <time>, and *y* is <mass>, then the *slope* must be in units of <mass>/<time>.

The control expression for the *line_rate* control might be as follows:

```
[ date = control: line_rate(); ]
```

The *line_rate* control might appear in a line_chart layout named *eff_line_chart.lyt*. The layout file *eff_line_chart.lyt* and its corresponding worksheet file are shown below.

```
// Layout
line_chart eff_line_chart
  worksheet: eff_line_chart;
  [ date = control: line_rate(); ]
  [ value = ; ]
  y: date value rate;
  visible: false;

//Worksheet
replicating eff_line_chart (Resource_Plan rp)
  [ head = "Eff Line Chart" ]
  [ pp = rp.efficiency_profile(rp.owner.owner
            .horizon); ]
  [ date = pp.date; ]
  [ qty = pp.value * 1.0; ]
  [ value = qty; ]
  [ rate = pp.rate; ]
```

4.1.3.6.1 Associated Actions

The following default actions are invoked from the *line_rate* control:

- Select
- Menu

4.1.4 list_bar

The *list_bar* control is a bar_chart control that is displayed within a bar_chart layout. A cell displayed with a *list_bar* control is expected to contain a list of values. All values in the list are stacked to form a single bar within the chart.

The data for *list_bar* comes across in a vector of values. Since there is only one control, things such as bar color are had to determine. Bar_color and tool_tip legend are determined as follows:

- If a *bar_color* property exists, then it is parsed and stored in a vector of colors. For example,

  ```
  [axis: time_buckets(time, res_name); ]
  [list_bar(capacity, product); property: bar_color =
     "blue, red"; ]
  ```

  This is different from the bar control in that it takes a comma-delimited list of colors 1-to-1 with the number of parameters in the *list_bar* signature. So, capacity shall be blue and product shall be red. These bars shall be stacked upon one another for each resource in each time bucket. If only one color is specified, then all bars in the *list_bar* get that color. If no color is specified as a property, then the default is used (black).
- The *tool_tip* string is derived from the title_attribute of the axis chart element of the chart_value. Only one of title_attribute exists and it happens to be the first bar's title. Therefore, from the example above for bar_color, the *tool_tip* for capacity and product reads "Capacity". However, the user can create a legend layout under the bar chart layout to explain the bar contents.

4.1.4.0.1 Associated Actions

The following default actions are invoked from the *list_bar* control:

- Select
- Menu

4.1.4.1 map_connect

map_connect is a map_chart control used to connect different nodes within a map chart. *map_connect* only appears within the context of a map_chart layout. The control expression for *map_connect* might be the following:

```
[ A1 = unique_id; control: map_connect; action:......; ]
```

The *map_connect* control might appear in a map chart named *bom_buf_buf_connects.lyt*. The layout file *bom_buf_connects.lyt* and its corresponding worksheet file are shown below.

```
// Layout:
axis_cross bom_buf_buf_connects
{
 worksheet: bom_buf_buf_connects;
 sparse: false;

 Y: A3 A2 C2 D1 D2 E1;
 [ A3 = control: map_connect; ]
}

//Worksheet:
replicating bom_buf_buf_connects (Item item)
{
 height: 3;
 width: 4;
 [ A1 buffer = item.buffers; ]
 [ A4 = recurse(buffer, for_each(#.producing_operation
                 .all_consume_flows,#.buffers))
                 .filter(#.flow_policy != "INFINITE"; ]
 [ A4.title = "A4"; ]
 [ C1 = for_each(A4.producing_operation.all_consume_flows,
                 #.buffer); ]
 [ C1.title = "From Buffer"; ]
 [ D1 = do(A4, 2); ]
 [ D1.title = "Arrow"; ]
 [ D2. = "solid"; ]
 [ D2.title = "dashes"; ]
 [ E1 = do(A4, "Yellow"); ]
 [ E1.title = "Color"; ]
 [ A2 = A4.id; ]
 [ A2.title = "To Buffer ID"; ]
 [ C2 = C1.id; ]
 [ C2.title = "From Buffer ID"; ]
 [ A3 = C1.current_index + 1; ]
 [ A3.title = "A3"; ]
}
```


4.1.4.1.1 Associated Actions

The following default actions are invoked from the *map_connect* control:

- Select
- Menu

4.1.4.2 map_node

map_node is a map_chart control used to display different nodes within a map chart. *map_node* only appears within the context of a map_chart layout. The control expression for *map_node* might be as follows:

```
[ A1 = unique_id; control: map_node; action:......; ]
```

The *map_node* control may appear in a map chart named *bom_buffer_nodes.lyt*. The layout file *bom_buffer_nodes.lyt* and its corresponding worksheet file are shown below.

```
//Layout
axis_cross bom_buffer_nodes
{
   worksheet: bom_buffer_nodes;
   Y: B1 C1 D1 E1 F1 G1 K1 J1 M1 H4;
   [ B1 = control: map_node; action: drag =
           display_report("buffer_editor", A1);
   action: MENU = echo("control menu:");
   action: dblclick = display_report("buffer_map", A1);
   action: select = 0; ]
}

//Worksheet:
replicating bom_buffer_nodes (Item item)
{
   height: 1;
   width: 11;
   [ A1 buffer = item.buffers; ]
   [ A2 = recurse(buffer,for_each(#.producing_operation
          .all_consume_flows, #.buffers)); ]
   [ A2.title = "Buffer"; ]
   [ B1 = A2.id; ]
   [ B1.title = "id"; ]
   [ C1 = A2.location.id; ]
   [ C1.title = "Parent ID"; ]
   [ D1 = do(A1, 8.0; ]
   [ D1.title = "x"; ]
   [ E1 = 10.0; ]
   [ E1.title = "y"; ]
   [ F1 = do(A1, 4.0); ]
   [ F1.title = "w"; ]
   [ G1 = do(A1, 4.0); ]
   [ G1.title = "h"; ]
   [ H1 = item.owner.owner; ]
   [ H2 = H1.plans.find("Abrasives Actual Plan"); ]
   [ H3 = H2.site_plans.find(item.owner); ]
```



```
[ H4 = H3.buffer_plans.find(A2).on_hand(H3.owner
        .horizon.start).number)*0.01;]
[ H1.title = "Qty1"; ]
[ J1 = "triangle,Buffer"; ]
[ J1.title = "Shape"; ]
[ K1 = left(A2.name, 16); ]
[ K1.title = "label"; ]
[ M1 = "Magenta, Buffer"; ]
[ M1.title = "Color"; ]
```

4.1.4.2.1 Associated Actions

The following default actions are invoked from the *map_node* control:

- Select
- Menu

4.1.4.3 time_buckets

The *time_buckets* control is a bar_chart control which specifies the cell values used to label the *x* axis. The *time_buckets* control is always used in conjunction with the *axis* property. The control expression for *time_buckets* might be as follows:

```
[ axis: time_buckets(B2); ]
```

The *time_buckets* control might appear in a bar_chart layout named *revenue_cost_bar_chart.lyt*. The layout file *revenue_cost_bar_chart.lyt* and its corresponding worksheet file are shown below:

```
//Layout
bar_chart revenue_cost_bar_chart
worksheet: revenue_cost_bar_chart;

[ axis: time_buckets(B2); ]
[ bar(F1); style: Bar1; width: 30; ]
[ bar(F2); style: Bar2; width: 40; ]

//Worksheet
replicating revenue_cost_bar_chart (Plan plan,
                           List[Date_Range] bucket_list)

[ head = "Revenue-Cost Bar Chart"; ];

[ B1 dates = bucket_list; ];
[ B2 dates = dates. start; ];
[ F1 cost = plan.site_plans.filter (#.site.managed)
                           .for_each(#.supply_request)
                           .filter(within(#.accepted,
                           dates)).for_each
                           (#.delivery_requests)
                           .for_each(#.delivery_promise)
                           .for_each(#delivery_price).sum; ];
[ F1.title = "Cost"; ];
[ F2 revenue = plan.site_plans.filter (#.site.managed)
                           .for_each(#.supply_request)
                           .fiter(within(#.accepted, dates))
                           .for_each(#.delivery_requests)
                           .for_each(#.delivery_promise)
                           .for_each(#delivery_price).sum; ];
[ F2.title = "Revenue"; ];
```


4.1.5 Table Controls

Table controls are controls used in table layouts. Table controls can only be used in table layouts. They are as follows:

- *percentage_bar*

4.1.5.1 percentage_bar

percentage_bar is a table control which displays a small graphic to represent the numeric value of the cell. Cell values which display using this control are expected to range from 0 to 100.

4.1.5.1.1 Associated Actions

The following default actions are invoked from the *percentage_bar* control:

- Select
- Menu

4.1.6 Text Controls

Text controls can be used in almost all layouts. Many text controls allow text to be inserted. Others restrict what can be typed or may invoke validation rules on the contents. Others are uneditable. The following controls are categorized as text controls:

- *button*
- *checkbox*
- *combo_popdown*
- *general*
- *image_control*
- *indented_general*
- *label*
- *layout*
- *menu_item*
- *menu_radio_item*
- *outline_general*
- *radio_button*
- *separator*
- *slider*
- *spinner*
- *submenu*
- *tool_button*
- *update_tool_button*

4.1.6.1 button

A *button* control displays as a button with a label that is associated with an action when selected. The control expression for the leftmost button would be the following:

```
[A1 = "Gantt Chart"; style: Button_Bar; control: button; ]
```

The first parameter in the expression is the label for the button.

The entire set of *button* controls might appear in an axis_cross layout named *menu_buttons.lyt*. The layout file *menu_buttons.lyt* and its corresponding worksheet file are shown below.

```
axis_cross menu_buttons ()

worksheet ()
  [A1 = "Gantt Chart"; style: Button_Bar; control: button;
       action: select = do (show("res_gantt_chart"),
       hide("res_chart1"), hide("res_chart2")); ];
  [ A2 = "Actual Load"; style: Button_Bar; control: button;
       action: select = do ( hide("res_gantt_chart"),
       show("res_chart1"), hide("res_chart2")); ];
  [ A3 = "Std Load";  style: Button_Bar; control: button;
       action: select = do ( hide("res_gantt_chart"),
       hide("res_chart1"), show("res_chart2")); ];
  [ A4 = "All Charts"; style: Button_Bar; control: button;
       action: select = do ( show("res_gantt_chart"),
       show("res_chart1"), show("res_chart2")); ];
  [ A5 = "Hide All"; style: Button_Bar; control: button;
       action: select = do ( hide("res_gantt_chart"),
       hide("res_chart1"), hide("res_chart2")); ];
 y: a1, a2, a3, a4, a5;
 default_rows: 1;
 visible: true;
 no_scroll: true;
```

4.1.6.1.1 Parameters

An alternate *button* control may be implemented for use in toolbars by using the following expression:

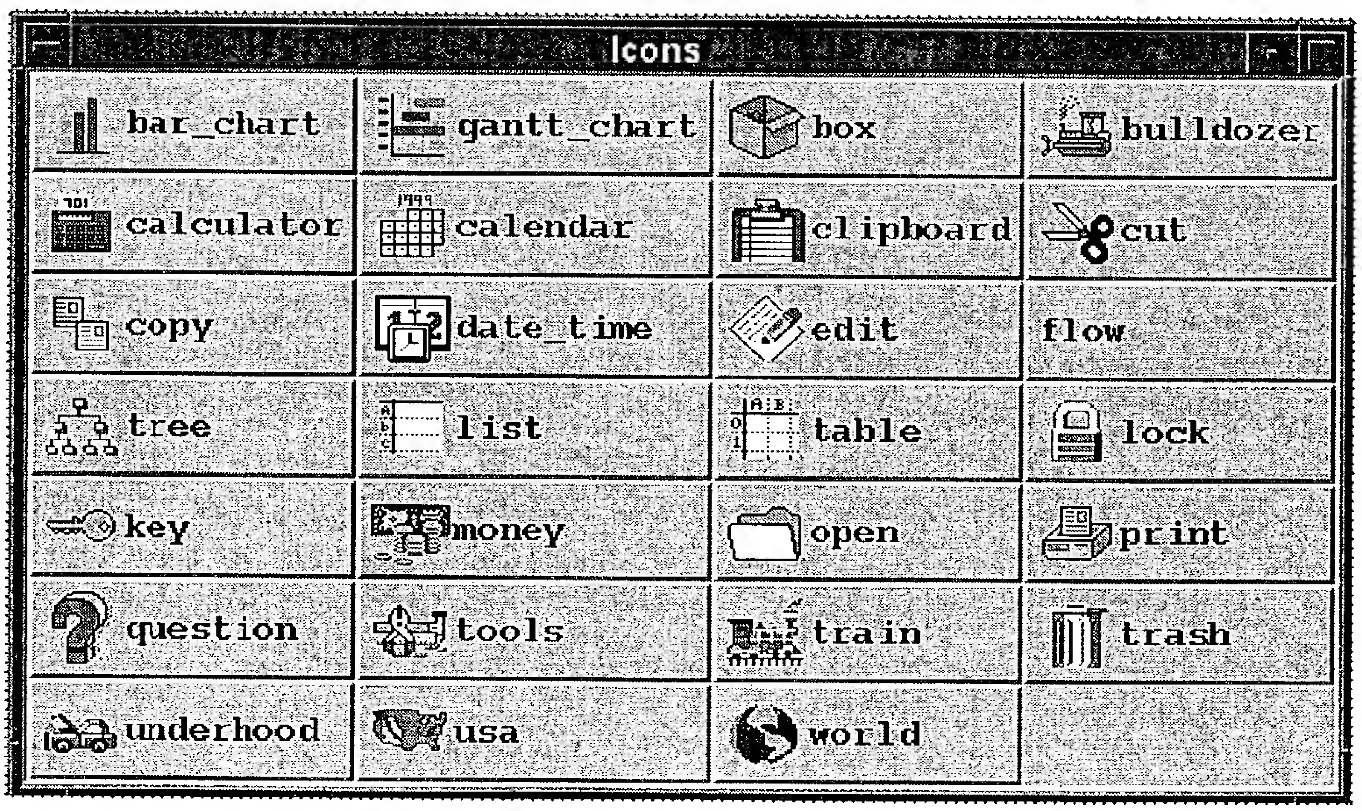
```
button(String image_name, bool vertical_p,
       bool center_p);
```

The button label, if any, comes from the cell value.

Table 8: Parameters

Parameter	Description
image_name	One of the defined images in FIGURE 22.
vertical_p	When true, this parameter means to place the label below the icon, else the label is to the right of the image.
center_p	When true, the parameter means the icon and label should be centered in both dimensions within the space available.

FIGURE 22 Visual Resource Buttons



4.1.6.1.2 Associated Actions

The following default action is invoked from the *button* control:

- Select

4.1.6.2 checkbox

A *checkbox* control displays as a two-state toggle. There are three ways in which a control can be written, each format affecting the way the title of the *checkbox* displays. The control expressions for *checkbox* are as follows:

- *checkbox ( )* - takes the title string from the default logical specification, and changes it to match the current toggle state. The default logical specification is found in the format file.
- *checkbox (String)* - uses *String* in double quotes as the checkbox title. Specify *("")* if a title is not to be displayed.
- *checkbox (String, String)* - uses *String, String* as true and false title strings, which change to match the current toggle state. *String, String* is positional; the first text string is for true, and the second is for false. When the toggle is depressed (true), then the first *String* is displayed. When the toggle is not depressed (false), then the second *String* is displayed.

4.1.6.2.1 Confirm

A checkbox automatically confirms (completes the edit by sending to the engine) when the control is selected by the mouse. That is, the following expression does not need to be added to every checkbox to get it to react more than just visually to the button push:

```
action: select = dispatch("confirm")
```


The *checkbox* control might appear in a spreadsheet layout named *style_colors.lyt*. The layout file *style_colors.lyt* and its corresponding worksheet file are shown below.

```
spreadsheet sed_colors(style)

worksheet (Style style)
  [A1 = "Invert?"; Style: Row_Title; Width: 170;];
  [B1 = style.invert; control: checkbox();
  [C1 = style.property_defined("invert");
     control: checkbox();  Width: 85;

  [A2 = "Foreground"; Style: Row_Title; Width: 170;];
  [B2 = style.foreground_color; Width: 170;];
   action: report = set_cell(get_color(gui_string
     ("relative", 0, 0)));];
  [C2 = style.property_defined("foreground_color");
     control: checkbox(); Width: 85;]

  [A3 = "Background"; Style: Row_Title; Width: 170;];
  [B3 = style.background_color; Width: 170;];
   action: report = set_cell(get_color(gui_string
     ("relative", 0, 0)));];
  [C3 = style.property_defined("background_color");
   control: checkbox(); Width: 85;]
```

4.1.6.2.2 Associated Actions

The following actions can be invoked from the *checkbox* control. Note that the functionality attached with these actions is not typical of the *checkbox* control. Note also that the *confirm* action OR the *select* action can be specified, but not both.

- Confirm
- Select

4.1.6.3 combo_popdown

A *combo_popdown* displays as a combo-box with a default value, and a pop-down list of alternatives. The control expression for this *combo_popdown* would be as follows:

```
[ A2 = vars.kind; control: combo_popdown; ]
```

When the down-arrow is pressed, the options of VARIABLE and CONSTANT are visible. VARIABLE is the default, as defined by OIL for the basic type of Variable_Kind.

4.1.6.3.1 Confirm

The *combo_popdown* automatically confirms (completes the edit by sending to the engine) when the control is selected by the mouse. That is, when an item is selected from a combo's menu, it is not necessary to press <Return> for the change to take effect.

The *combo_popdown* control might appear in an axis_cross layout named *site_plan_editor.lyt*, with the popdown consisting of options LINK, SUPPLIER, and CUSTOMER. The layout file *site_plan_editor.lyt* and its corresponding worksheet file are shown below.

```
axis_cross site_plan_editor

worksheet: site_plan_editor
     Y: F1 G1 D1

     [ F1 = control: combo_popdown; ]
     [ G1 = control: checkbox; ]
     [ D1 = action: report = dispatch("model_edit"); ]

//Worksheet

replicating site_plan_editor (Site_Plan site_plan)
     [ B1 site = site_plan.site; ]
     [ C1 plan_description = site_plan.description; ]
     [ C2 site_description = site_plan.site.description; ]
     [ D1 = site_plan.organization_plan; ]
     [ E1 plan = site_plan.owner; ]
     [ F1 = site.role; ]
     [ G1 = site.managed; ]
```

This layout with the *combo_popdown* control is shown when the *Supply Chain Editor* button control is selected.

4.1.6.3.2 Layout

All layouts used in a report must be named as a layout in the *.rpt* file. Therefore, even if the layout for the *combo_popdown* does not appear directly on the report, but is contained within another layout of the report, it must be listed in the *.rpt* file. The following code fragments illustrate this more completely.

```
//file my_report.rpt
my_report (...some parameters...)
{
  ...
  layout: layout_that_actually_shows_in_the_report();
  layout: nonphysical_datatable_types();
  ...
}

//file layout_that_actually_shows_in_the_report.lyt
spreadsheet layout_that_actually_shows_in_the_report
{
  worksheet: layout_that_actually_shows_in_the_report
  ...
  [ E4 = control:combo_popdown
        ("nonphysical_datatable_types", TRUE); ]
  ...
}

//file layout_that_actually_shows_in_the_report.wrk
normal layout_that_actually_shows_in_the_report()
{
  ...
  [ E4 = values("Data_Table_type_Enum")
        .filter(#! = "PHYSICAL").first; ]
  //the line above computes an initial value for the
        combo_box
  ...
}

//file nonphysical_datatable_types.lyt
axis_cross nonphysical_datatable_types
    {
    worksheet: nonphysical_datatable_types;
     sort: A1;
     y: A1;
}
```

```
//file nonphysical_datatable_types.wrk
replicating nonphysical_datatable_types()
{
  [ A1 = values("Data_Table_type_Enum")
           .filter(#! = "PHYSICAL"); ]
  [ A1.title = ""; ]
}
```

4.1.6.3.3 Associated Actions

The following actions can be invoked from the *combo_popdown* control. Note that the functionality attached with these actions is not typical of the *combo_popdown* control. Note also that the *confirm* action OR the *select* action can be specified but not both.

- Confirm
- Select

4.1.6.4 general

general is a control that displays as a string. It is typically used in formatting the date and time styles within a layout. If the cell containing the control has a command whose name matches an icon name, then the icon is displayed to the right of the string. Clicking on the icon invokes the command. Common icon/actions are as follows:

- combo_popdown choices
- toggle-through menu choices
- including more information on a report
- *Report* buttons which open the report that is the next logical in the flow

The control expression for a *general* control would be the following:

```
[ C1 = control: general("short"); ]
```

Note: For spreadsheet and axis cross layouts, the default control is *general.* If the user does not specify otherwise, the *general* control is automatically set for that cell.

The *general* control may appear in a layout named *active_problem_list.lyt*. The layout file *active_problem_list.lyt* and its corresponding worksheet file are shown below.

```
//Layout:
//
axis_cross active_problem_list
{
   worksheet: active_problem_list;
   sort: C1;
   sparse: false;
   Y: L1 A2 C1 D1 E1 F1 ("Z1" G1 H1 I1 J1 K1 K2) M1;
   [ C1 = control: general("short"); ]
   [ F1 = action: more = display_report("more", F1 &
                  " Problem", A3); ]
   [ G1 = action: report = dispatch("model_edit"); ]
   [ H1 = action: report = dispatch("model_edit"); ]
   [ I1 = action: report = dispatch("model_edit"); ]/
   [ J1 = action: report = dispatch("model_edit"); ]
   [ K1 = action: report = dispatch("model_edit"); ]
   [ K2 = action: report = dispatch("model_edit"); ]
   [ G1.title = style: Row_Title; ]
   [ H1.title = style: Row_Title; ]
   [ I1.title = style: Row_Title; ]
   [ J1.title = style: Row_Title; ]
   [ K1.title = style: Row_Title; ]
   [ K2.title = style: Row_Title; ]
   [ L1 = control: button();
                  action: select = A3.resolve(A1.owner); ]
   [ A3 = action: report = dispatch("model_edit"); ]
   [ M1 = protected: true; ]
}
```



```
//Worksheet
//
replicating active_problem_list (List[Active_Problem]
                                  active_problems)
{
    [ A1 a_problem = active_problems; ]
    [ A1.title = "Problems"; ]
    [ A2 = a_problem.focus; ]
    [ A3 problem = a_problem.problem; ]
    [ B1 = problem.description; ]
    [ C1 = problem.dates; ]
    [ C1.title = "Plan Dates"; ]
    [ D1 = problem.cost; ]
    [ E1 = problem.interaction; ]
    [ F1 = problem.category; ]
    [ G1 = problem.resource_plan; ]
    [ H1 = problem.buffer_plan; ]
    [ I1 = problem.operation_plan; ]
    [ I1.title = "Operation Plan"; ]
    [ J1 = problem.operation.state; ]
    [ K1 = problem.request; ]
    [ K2 = problem.item_request; ]
    [ L1 = do(problem, "Resolve"); ]
    [ L1.title = ""; ]
    [ M1 = problem.last_change; ]
    [ M1.title = "Detected"; ]
    [ Z1 = ""; ]
    [Z1.title = "Details"; ]
}
```

4.1.6.4.1 Associated Actions

The following actions can be invoked from the *general* control. Note that the functionality attached with these actions is not typical of the *general* control.

- Choose
- Confirm (only if the cell is editable)
- Map
- Menu
- More
- Report
- Select

4.1.6.5 image_general

image_general (`image_general (String, String, Integer)`) displays as a string and an image (icon). Unlike using the general control, the *image_general* control cell will NOT draw an icon to the right of the string if the cell containing the control has a command whose name matches an icon name. Instead, the second string parameter is the name of an image. This image shall be drawn next to the string. The third parameter controls the placement of the image: 0=Left, 1=Right, 2=Top, 3-Bottom. In all cases, the image and text will be centered with respect to one another.

The control expression for *image_general* is:

```
[ B2 = control: image_general("default", "scp_logo", 0); ]
```

The *image_general* control appears in the layout named *main_header.lyt*. This layout file and its corresponding worksheet file are shown below:

```
// Layout:
spreadsheet main_header
{
 worksheet: main_header;
 style: Button;
 action: show_report = domains.find(domain).activities
      .find(activity).display;
 action: ok = dispatch("show_report");
 [ A1 = width: 5; height: 10; ]
 [ B2 = control: image_general("default", "scp_logo", 0); ]
 [ C2 = width: 60; ]
 [ D2 = control: button(); top_inset: 15; bottom_inset: 15;
      action: select = dispatch("show_report"); ]
 [ E2 = width: 140; ]
 [ F2 = control: image_general("default", "i2_small", 0); ]
 [ G3 = width: 5; height: 10; ]
}

// Worksheet:
normal main_header (String domain, String activity,
                    String io)
{
   [ A1 = ""; ]
   [ B2 = ""; ]
   [ C2 = ""; ]
   [ D2 = "    Display Report    "; ]
   [ E2 = ""; ]
   [ F2 = ""; ]
   [ G3 = ""; ]
}
```


4.1.6.6 indented_general

indented_general is a control used to create a list which is indented by its depth. The notion of *depth* in a list arises from the use of the recurse function. The recurse function assigns depth based on the depth of the recursion. The difference between *indented_general* and *outline_general* is that *outline_general* has connectors between items in the list. The control expression would be as follows:

```
[ B1 = control: indented_general("", 18); ]
```

The *indented_general* control might appear in a layout named *product_forecasts.lyt*. The layout file *product_forecasts.lyt* and its corresponding worksheet file are shown below:

```
//Layout:
//
axis_cross product_forecasts
{
worksheet: sub_forecasts;
X: C2;
Y: B1 CROSS;
CROSS: E3 E4 E6 E8

[ B1 = control: indented_general("", 18); ]
[ B3 = control: checkbox(); ]
[ C1 = control: general("short"); ]
[ C2 = control: general("short"); ]
}
```

4.1.6.6.1 Associated Actions

The following actions can be invoked from the *indented_general* control. **Note:** The functionality attached with these actions is not typical of the *indented_general* control.

- Choose
- Map
- Menu
- More
- Report

4.1.6.7 label

label is a control used as a string that cannot receive focus. Therefore, it can be used to show strings in reports that are not supposed to be edited.

4.1.6.8 layout

The *layout* control displays a layout of any type within a cell. If the *x* and *y* layout properties are not present, then the cells are sized and positioned automatically. Since this is true for *layout* controls, side-by-side placement can be obtained by putting the controls in adjacent cells in the parent spreadsheet.

The control expression for the main_header layout would be as follows:

```
[ A1 = control: layout("main_header"); ]
```

The entire set of *layout* controls for the main report might appear in a layout named *main_body.lyt.* The layout file *main_body.lyt* and its corresponding worksheet file are shown below.

```
//Layout
spreadsheet main_body
worksheet: main_body;
style: Button;
no_scroll: true;
[ A1 = control: layout("main_title"); ]
[ A2 = control: button("juggler_large", true, true);
        action: select =.display_report("about_scp") ]
[ B1 = control: layout("main_header"); ]
[ B2 = control: layout("main_content"); ]

//Worksheet
normal main_body ()
[ A1 = ""; ]
[ B1 = ""; ]
[ A2 = ""; ]
[ B2 = ""; ]
```

4.1.6.9 menu_item

A *menu_item* control appears as an item in a menu, menu_bar, or submenu. It has actions associated with it that are invoked when the menu item is selected. The example below shows the **Close** *menu_item* control. This menu item exists in the cascaded *submenu* control of the *File* menu layout in the main menubar layout. The control expression for the **Close** *menu_item* would be as follows:

```
[ E1 = control: menu_item("close"); ]
```

The *"close"* in the above example references the command in the corresponding worksheet which specifies to *Close* the window.

The *menu_item* control might appear in a menu layout named *res_plan_file_menu.lyt* accessed by a submenu layout for a menubar layout named *res_plan_file_menu.lyt.* The layout file *res_plan_file_menu.lyt* and its corresponding worksheet file are shown below.

```
//Layout
menu res_plan_main_menu
worksheet: res_plan_main_menu;
     [ A1 = control: menu_item("save_res_plan_main_menu; ]
     [ B1 = control: menu_item
                    ("import_res_plan_main_menu"); ]
     [ D1 = control: menu_item("update"); ]
     [ E1 = control: menu_item("close"); ]

//Worksheet
normal res_plan_main_menu ()
height: 1;
width: 6;
     [ A1 = "Save"; ]
     [ B1 = "Import Res_Plan_Main"; ]
     [ C1 = "Export Res_Plan_Main"; ]
     [ D1 = "Update Report"; ]
     [ E1 = "Close _Window^w"; ]
```

4.1.6.9.1 Associated Actions

There are no actions supported for the *menu_item* control. The associated action is specified through the argument list.

4.1.6.10 menu_radio_item

menu_radio_item is a control which displays as an exclusive menu toggle item, either in a menubar or a menu. Only one *menu_radio_item* may be selected within a set of such items bound by separators inside a menu. The *menu_radio_item* control indicates which item is chosen from the menu by displaying a checkmark next to that item. The control expression would be as follows:

```
[ = "Whole Horizon";
        control: menu_radio_item("horizon_bucket"); ];
```

The *menu_radio_item* control appears in the layout *buckets_menu.lyt* for the example above. The layout file *buckets_menu.lyt* and its corresponding worksheet file are shown below.

```
//Layout:
menu buckets-menu
{
     worksheet: buckets_menu;
}

//Worksheet:
normal buckets_menu ()
{
  [ = "Whole Horizon";control:
menu_radio_item("horizon_bucket"); ];
  [ = "Quarters";     control:
menu_radio_item("quarter_buckets"); ];
  [ = "Months";       control:
menu_radio_item("month_buckets"); ];
  [ = "Weeks_months"; control:
menu_radio_item("week_month_buckets"); ];
  [ = "Weeks";        control:
menu_radio_item("week_buckets"); ];
  [ = "Days-Weeks-Months";
           control:
menu_radio_item("day_week_month_buckets"); ];
  [ = "Custom";       control:
menu_radio_item("custom_buckets"); ];1
}
```

4.1.6.10.1 Associated Actions

There are no actions supported for the *menu_radio_item* control. The associated action is specified via the argument list.

4.1.6.11 outline_general

outline_general is a control which displays a list with collapse/expand indicators. It graphically portrays the tree structure and provides little boxes containing "+" or "-" on nodes with children to perform collapse/expand functions. A left click toggles the expansion state. If currently collapsed, all immediate children are expanded. If currently expanded, all children are collapsed. A shift-left click always expands all children.

There are two versions of this control:

```
outline_general (String)
outline_general (String, Integer)
```

The *outline_general(String, Integer)* version of this control takes a maximum initial recursion level parameter (Integer) which allows the user to control whether or not all entries are displayed in the expanded format. Otherwise, both versions of *outline_general* behave identically. The control expression would be as follows:

```
[ A1 = control: outline_general("L");
        style: Value_Cell_Outline;
        action: report = dispatch("model_edit"); ]
```

The *outline_general* control might appear in a layout named *site_plan_list.lyt*. The layout file *site_plan_list.lyt* and its corresponding worksheet file are shown below:

```
//Layout:
//
axis_cross site_plan_list
{
   worksheet: site_plan_list;
   Y: A1 F1 B2;

   [ A1 = control: outline_general(""); style:
Value_Cell_Outline; action: report =
   dispatch("model_edit"); ]
   [ F1 = control: combo_popdown; ]
   [ B2 = control: checkbox(); ]
}
```



```
//Worksheet:
//
replicating site_plan_list (List[Site_Plan] site_plans)
{
   height: 2;
   width: 11;
   [ A1 site_plan = site_plans.recurse(#.members); ]
   [ B1 site = site_plan.site; ]
   [ C1 plan_description = site_plan.description; ]
   [ C2 site_description = site_plan.site.description; ]
   [ D1 = site_plan.organization_plan;}
   [ E1 plan = site_plan.owner; ]
   [ F1 = site_plan.role;]
   [ G1 = site_plan.members;]
   [ H1 = site_plan.resource_plans; ]
   [ I1 = site_plans.buffer_plans; ]
   [ J1 = site_plan.operation_plans; ]
   [ K1 = site_plan.requests; ]
   [ O1 = site_plan.operation_states; ]
   [ L1 = do(site_plan, ""); ]
   [ N1 problem_count = do(site_plan, 0); ]
   [ B2 = site.managed; ]
}
```

4.1.6.11.1 Associated Actions

The following actions can be invoked from the *outline_general* control. **Note:** The functionality attached with these actions is not typical of the *outline_general* control.

- Choose
- Map
- Menu
- More
- Report

4.1.6.12 radio_button

radio_button is a control which displays as an exclusive toggle button with a title string. If the empty string is specified, no title strings will be displayed. Its initial state is taken from the value of the cell. One control expression would be as follows:

```
[ E3 = control: radio_button(""); ]
```

The *radio_button* control might appear in a layout named *supply_chain_list.lyt*. The layout file *supply_chain_list.lyt* and its corresponding worksheet file are shown below:

```
//Layout:
axis_cross supply_chain_list
{
   worksheet supply_chain_list;
   sort: A1 E1;
   Y: E3 A1 E1;

   [ E3 = control: radio_button(""); ]
   [ A1 = action: report = display_report
            ("supply_chain_editor", supply_chain); ]
   [ E1 = action: map = display_report("plan_map", E1);
           action: report = display_report
             ("plan_editor"), E1); ]
}
```



```
//Worksheet:
replicating supply_chain_list (List[Supply_Chain]
                              supply_chains, Plan chosen_plan)
{
   [ A1 supply_chain = supply_chains; ]
   [ A1.title = "Supply Chains"; ]
   [ A2 = supply_chain.name; ]
   [ B1 = supply_chain.description; ]
   [ C1 = supply_chain.sites; ]
   [ C2 = "Sites (" & supply_chain.sites.count.string &")"; ]
   [ C2.title = "Sites"; ]
   [ D1 = supply_chain.sellers; ]
   [ D2 = "Sellers(" & supply_chain.sellers.count
                              .string &")"; ]
   [ D2.title = "Sellers"; ]
   [ E1 plans = supply_chain.plans; ]
   [ E2 = "Plans (" & supply_chain.plans.count.string &")"; ]
   [ E2.title = "Plans"; ]
   [ E3 chosen = (chosen_plan == E1), = set_variable
                              (chosen_plan, E1); ]
   [ E3.title = ""; ]
```

4.1.6.12.1 Associated Actions

The following action is invoked from the *radio_button* control:

- Select (if the user sets a variable)

The following action can be invoked from the *radio_button* control. **Note:** The functionality attached with this action is not typical of the *radio_button* control.

- Confirm

4.1.6.13 separator

A *separator* control appears as a horizontal line in a menu or submenu. It breaks the menu or submenu up into sections. It allows you to organize a menu into logical sections. It also makes menus easier to use. You can put a *separator* line before a command such as *Quit*, making it less likely that the command is selected unintentionally. The *separator* line is not grayed, but it is not selectable. The control expression for a *separator* is as follows:

```
[ E1 = control: menu_item("save_as"); ]
[ F1 = control: separator(); ]
[ G1 = control: menu_item("import"); ]
```

Note: The *separator* between the *Save As* and *Import* menu items.

The *separator* control might appear in a layout named *editor_file_menu.lyt*. The layout file *editor_file_menu.lyt* and its corresponding worksheet file are shown below.

```
//Layout:
menu editor_file_menu
{
worksheet: editor_file_menu;
visible: FALSE;
[ A1 = control: menu_item("Start"); ]
[ B1 = control: menu_item("open"); ]
[ C1 = control: menu_item("revert"); ]
[ D1 = control: separator(); ]
[ E1 = control: menu_item("oil_test"); ]
[ F1 = control: menu_item("oil_commit"); ]
[ G1 = control: menu_item("oil_save_as"); ]
[ H1 = control: separator(); ]
[ J1 = control: menu_item("exit"); ]
}
```



```
//Worksheet:
normal editor_file_menu ()
{
 height: 1;
 width: 19;
 [ A1 = "_New"; ]
 [ B1 = "_Open"; ]
 [ C1 = "_Revert"; ]
 [ D1 = ""; ];
 [ E1 = "_Apply Changes^m"; ]
 [ F1 = "_Save Changes"; ]
 [ G1 = "_Save Changes As..."; ]
 [ H1 = ""; ];
 [ I1 = copy_cells("base_file_menu"); ]
 [ J1 = "Exit"; ]
}
```

4.1.6.14 slider

A *slider* control appears as a slider. The *slider* allows a user to select from a range of values. The user can slide the *slider* to change the underlying value of the *slider*. The *slider* has a range of values that delimit its movement.

To illustrate, consider the traditional text control in a cell that provides the traditional spreadsheet cell behavior of textually editing the value of the cell. That control can be replaced in a layout by a *slider* control (if the cell type is numeric). The *slider* gives a graphic handle that slides in a groove that is indexed with the range of possible values. The user can change the value by sliding the handle via interaction with a mouse (or other input device).

The *slider* control might appear in a layout named *active_goal_list.lyt.* The layout file *active_goal_list.lyt* and its corresponding worksheet file are shown below:

```
//Layout
axis_cross active_goal_list
worksheet: active_goal_list;
sort: E1 B1;
Y: E1 E2 B1 D2 D1 C2;

[ E2 = control: slider(0, 1); ]
[ B1 = control: combo_popdown; ]
[ C2.title = style: Row_Title; ]
[ C3.title = style: Row_Title; ]
[ C4.title = style: Row_Title; ]
[ C5.title = style: Row_Title; ]

//Worksheet
replicating active_goal_list (List[Active_Goal] goals)
[ A1 goal = goals; ]
[ A1.title = "Goals"; ]
[ B1 = goal.strategy_goal.goal; ]
[ C2 = goal.total_interaction; ]
[ C3 = goal.problem_count; ]
[ C4 = goal.total_lateness; ]
[ C5 = goal.total_shortness; ]
[ D1 = goal.adjusted_value; ]
[ D2 = goal.strategy_goal.adjusted_target; ]
[ E1 focus = do(goal.focus_value, percentage(1)); ]
[ E2 = focus; ]
[ E2.title = ""; ]
```


4.1.6.14.1 Associated Actions

The following actions can be invoked from the *slider* control. Note that the functionality attached with these actions is not typical of the *slider* control. Note also that the *confirm* action OR the *select* action can be specified but not both.

- Confirm
- Select

4.1.6.15 spinner

The *spinner* control allows a user to select from a list of values. The user can click on arrow keys to select the next or previous value in the list. The *spinner* has a range of counters that delimit it movement. Each counter indicated a different item in the list. The *spinner* works by using the arrows on the *spinner* and by editing the number display box on the control and pressing the <Return> key. The following expressions present one way to implement a *spinner*.

```
[ D1 =control: spinner(1, 1000000); ]
```

The *spinner* returns the last accepted value not the string seen in the cell.

The *spinner* control might appear in a layout named *skill_list.lyt*. The layout file *skill_list.lyt* and its corresponding worksheet file are shown below:

```
//Layout
axis_cross skill_list

worksheet: skill_list;
Y: A1 E1 D2 C1;
sort: A1;

[ A1 = action: report = dispatch("model_edit"); ]
[ E1 = control: combo_popdown; ]
[ D1 = control: spinner(0,1000000); ]
```



```
//Worksheet
replicating skill_list (List[Skill] skills)
height: 2;
width: 9;
[ A1 skill = skills; ]
[ A1.title = "Skill"; ]
[ B1 = skill.name; ]
[ B1.title = "Name"; ]
[ C1 = skill.description; ]
[ C1.title = "Description"; ]
[ D1 = skill.resources; ]
[ D1.title = "D1"; ]
[ E1 = skill.selection; ]
[ E1.title = "Selection"; ]
[ F1 = skill.loading_operations; ]
[ F1.title = "Loading_Operations"; ]
[ G1 = skill.loading_buffers; ]
[G1.title = "Loading Buffers"; ]
[ H1 = skill.owner; ]
[ H1.title = "Site"; ]
[ I1 = skill.owner.owner; ]
[ I1.title = "Supply Chain"; ]
[ D2 resource_count = skill.resources.count; ]
[ F2 = skill.loading_operations.count.string &
             " loading operation" & if(skill
             .loading_operations.count == 1, "".”s"); ]
[ F2.title = "Loading Operations"; ]
[ G2 = skill.loading_buffers.count.string &
             "loading buffer" & if(skill.loading_buffers
             .count == 1, "".”s"_; ]
[ G2.title - "Loading Buffers"; ]
[ Z1 = ""; ]
[Z1.title = ""; ]
```

4.1.6.15.1 Associated Actions

The following actions can be invoked from the *spinner* control. **Note:** The functionality attached with these actions is not typical of the *spinner* control. Note also that the *confirm* action OR the *select* action can be specified but not both.

- Confirm
- Select

4.1.6.16 submenu

A *submenu* control appears as a submenu of another menu. The example below shows the *submenu* control for the *File* entry in the menubar layout. It has one argument which is the name of another menu layout to use as the submenu. *submenu* is used to organize menus. For example, the *Edit* menu might contain the submenu *OIL*, which might have the *Report*, *Layout*, and *Worksheet menu_items*. The control expression for the **File** *submenu* might be as follows:

```
[ A1 = control: submenu("editor_file_menu"); ]
```

The *submenu* control that displays the menu of file and edit submenu items might appear in a menubar layout named *full_sed_menubar.lyt*. The layout file *full_sed_menubar.lyt* and its corresponding worksheet file are shown below.

```
//Layout
menubar full_sed_menubar ()
 worksheet: fill_sed_menubar;
 style: Menubar;
   [ A1 = control: submenu("editor_file_menu"); ]
   [ B1 = control: submenu("editor_edit_menu"); ]

//Worksheet
normal full_sed_menubar ()
   height: 1;
   width: 2;
   [ A1 = "_File"; ]
   [ B1 = "_Edit"; ]
```


4.1.6.17 tool_button

The *tool_button* control displays as a button with an icon and a label string. When pressed, the *tool_button* will dispatch the specified action to the current focus cell. *tool_button* controls can only occur within a toolbar layout. The control expression for **Close this Report** *tool_button* would be as follows:

```
[ A1 = control: tool_button("close_report", "close",
    "Close this Report"); ]
```

The entire set of *tool_button* controls might appear in a toolbar layout named *scp_plan_toolbar.lyt.* The layout file *scp_plan_toolbar.lyt* and its corresponding worksheet file are shown below.

```
//Layout
toolbar scp_plan_toolbar
no_scroll: True;

[ A1 = control: tool_button("close_report", "close",
        "Close this Report"); ];
[ B1 = control: tool_button("freeze_report", "freeze",
        "Freeze this Report -- No Updates"); ];
[ C1 = control: tool_button("update_report", "update",
        "Update this Report"); ];
[ D1 = control: tool_button("update_all", "update_all",
        "Update All Reports"); ];
[ E1 = control: tool_button("print_report", "print",
        "Print this Report"); ];
[ F1 = width: 8; ];

//Worksheet
normal scp_plan_toolbar ()
[ A1 = ""; ];
[ B1 = ""; ];
[ C1 = ""; ];
[ D1 = ""; ];
[ E1 = ""; ];
[ F1 = ""; ];
```

4.1.6.17.1 Associated Actions

There are no actions supported for the *tool_button* control. The associated action is specified via the argument list.

4.1.6.18 update_tool_button

The *update_tool_button* displays as a button with an icon and a label string. When pressed, this button dispatches the specified action to the current focus cell. The *update_tool_button* control can only occur within a toolbar layout.

The *update_tool_button* also has an extra image name used to indicate when a report needs to be updated. When a value is changed within the report, the *update_tool_button* turns yellow. The control expression for the *update_tool_button* is as follows:

```
[C1 = control: update_tool_button("update_report",
   "update_needed", "update", "Update this Report"); ];
```

The *update_tool_button* control appears in the layout named *main_toolbar.lyt*. The layout file and its corresponding worksheet file are shown below:

```
//Layout:
toolbar main_toolbar
{
   worksheet: main_toolbar;
   no_scroll: True;
   [ A1 = control: tool_button("close_report", "close",
      "Close this Report"); ];
   [ B1 = control: tool_button("freeze_report", "freeze",
      "Freeze this Report -- No Updates"); ];
   [ C1 = control: update_tool_button("update_report",
      "update needed", "update", "Update this Report"); ];
    [ D1 = control: tool_button("update_all", "update_all",
      "Update All Reports"); ];
    [ E1 = control: tool_button("print_report", "print",
      "Print this Report"); ];
    [ F1 = width: 8; ];
    [ G1 = control: tool_button("save", "save",
      "Save the Rhythm SCP Model"); ];
    [ H1 = control: tool_button("import", "import",
      "Import External Data into Rhythm"); ];
    [ I1 = control: tool_button("export", "export",
      "Export Data from Rhythm"); ];
    [ J1 = width: 8;];
    [ K1 = control: tool_button("cut", "cut", "Cut"); ];
    [ L1 = control: tool_button("copy", "copy", "Copy"); ];
    [ M1 = control: tool_button("paste", "paste",
      "Paste"); ];
    [ N1 = width: 8; ];
```



```
    [ O1 = control: tool_button("help", help_on_cell",
       "Help on Cell"); ];
  }

//Worksheet
normal main_toolbar ()
[ A1 = ""; ];
[ B1 = ""; ];
[ C1 = ""; ];
[ D1 = ""; ];
[ E1 = ""; ];
[ F1 = ""; ];
[ G1 = ""; ];
[ H1 = ""; ];
[ I1 = ""; ];
[ J1 = ""; ];
[ K1 = ""; ];
[ L1 = ""; ];
[ M1 = ""; ];
[ N1 = ""; ];
[ O1 = ""; ];
```

4.2 Formats

Basic types (e.g. Integer) have *formats* associated with them. *Formats* specify how the values are expected to be structured on input and output, for instance, how many decimal places are to be displayed. *Formats* are specified in *.fmt* files. The specification syntax is shown below:

```
format format_name Type
  {
  specification: format_specification;
  {
```

where *format* and *specification* must be typed as shown and substitutions must be made for *format_name Type* and *format_specification*. The file-name of the *.fmt* file is *format_name_Type.fmt*.

4.2.1 Invoking Formats

There are several methods of invoking *formats*. Models are usually formatted according to the type of their key fields. Many controls (e.g. general) accept *format_name* as a parameter. If not given such a named format, each worksheet cell will look for the format named "default" corresponding to the type of its value. For an example, refer to the following files located in the *.../reports/basic* directory:

- *default_date.fmt* (for type Date)
- *default_logical.fmt* (for type Logical)

FIGURE 23 shows an example of using *format* to specify a special Number format.

FIGURE 23 Format Example

format in .fmt file

```
format special Number {
specification: n(####.00n) "special";
}
```

Using format in a layout

```
[ A1 = control: general("special"); ]
```

If *A1* contains *456*, it displays as *456.00special.* If *A1* contains *-10.34*, it displays as *(10.34)special.*

4.2.2 Available Formats

The following is a list of all available formats:

Format	Description
accounting_number	"n(#, ##0.00n)";
accounting_quantity	"n(#, ##0.00n) S3";
comma_number	"n-#, ###.##";
comma_quantity	"N-#, ###.## S3";
dd_date	"DD";
default_date	"YY-MM-DD hh:mm";
default_date_range	"YY-MM-DD hh:mm"; separator: " / ";
default_integer	"%d";
default_list	element_format: default delimiter: "," ; width: 100; truncated_format: "(+ {0} ...)";
default_logical	"No, Yes, False=0, True=1, 0=0, 1=1";
default_number	"*";
default_percentage	"n-##.##\"%\" ";
default_quantity	"n-##.## S3";
default_quantity_range	"n-##.## S3"; separator: "<";
default_restriction	"YY-MM-DD hh:mm";
default_string	
default_time	
dmy_hms_date	"DD-MM-YYYY hh:mm:ss";
full_list	width: 2000000;
hm_date	"hh:mm";

Format	Description
hms_date	“hh:mm:ss”;
hms_time	“hh:mm:ss”;
mm_dd_yy_date	“MM/DD/YY”;
month_date	“YY-MMM”;
newline_list	delimiter: “\n”; width: 99999;
short_date	“YY-MM-DD”;
short_date_range	“YY-MM-DD”; separator: “ / ”;
short_percentage	“n-##\”%\“ ”;

4.2.3 Special Notes on Controls and Format

There are a few items of special importance to mention regarding formats. They are listed below:

- Controls cannot be created using OIL
- Formats can be created using OIL, but the *Reload Report Definitions* menu option will not work (you must shut the engine down and restart).

4.3 Styles

Styles can be used to control the appearance of the cells by modifying their:

- colors
- fonts
- borders

Styles are specified in *.sty* files. The exact properties of a *style* that are applied depend on the control in which the *style* is used. The same *styles* can be used across many reports which ensures uniformity in the look and feel of reports.

4.3.1 Syntax for Specifying Style

Style requires a particular syntax within OIL. This syntax is shown below:

```
cell_style style_name {
  <property: value>
  ... }
```

where *cell_style* must be typed as shown and substitutions must be made for *style_name*, *property* and *value*. See *Title.sty* for an example.

4.3.2 Invoking Styles

Style can be used within the layout cell specification:

```
[ cell_id = style: style_name; ]
```

where *style* must be typed as shown and a substitution must be made for *cell_id* and *style_name*. This specifies a style for this layout cell only. It is also possible to specify a default style for an entire layout by adding a style declaration to the properties of the layout. For example,

```
spreadsheet x
{ ...
  style: style_name;
  ...
}
```

A layout cell's style will override the default style. (See *plan_context.lyt* for an example of the use of *style*.)

4.3.3 Available Styles

The following is a list of all available styles and their descriptions:

Style	Description
foreground_color	The name of the foreground color.
background_color	The name of the background color.
format	The format applied to the value string.
border_color	The name of the border color.
pattern	The name of the stipple pattern to use.
pattern_color	The name of the stipple pattern color to use.
h_align	The horizontal alignment type.
v_align	The vertical alignment type.
invert	The specification, if TRUE, to swap foreground and background colors.
font_family	The font family, such as arial, or courier.
font_style	The font style, such as normal, italic, or bold.

Style	Description
font_size	The font size.
border_style	The default border style. Possible values include: bare, single_line, double_line, wide_line, dash_line, wide_dash_line, shadow_in, wide_shadow_in, shadow_out, and wide_shadow_out.
border_top	The border style for the top line.
border_right	The border style for the right line.
border_left	The border style for the left line.
border_bottom	The border style for the bottom line.
width	The width of the cell area.
height	The height of the cell area.
disable	The specification to draw an item greyed-out and to ignore input events.
hide	The specification to make an item invisible.
protected	The specification, if TRUE, to make the control value uneditable.
modal	The specification, if TRUE, to make the report modal.

4.3.4 Applying Conditional Styles

The UI does not hard-code style names. Conditionally applicable styles have the keyword *condition*, as in the following example:

```
style Selected
{
  condition: selected;
  // colors, fonts, etc.
}
```

The accepted values for condition are SELECTED, EDITABLE, and FOCUS.

If a style file in your private reports area does not use the *condition* keyword, RHYTHM SCP will apply the style unconditionally to all cells.

4.3.5 Special Notes on Styles

There are a few items of special importance to mention regarding styles. They are listed below:

- Styles can be created using OIL, but the *Reload Report Definitions* menu option will not work (you must shut the engine down and restart).
- Appendix C contains UI design guidelines for applying controls, styles, etc.
- Appendix E lists the built-in images provided by OIL for use as icons, etc.

Section 5

Using Events, Actions, and Bindings

You are now going to learn to apply events, actions, and bindings to the basic report you learned to create in the first sections.

5.1 Events

A GUI *event* is basically a user gesture, such as clicking a mouse button or selecting a menu item. There are ways to associate any software behavior with any GUI *event*. This is done through the use of:

- actions
- key bindings
- `dispatch` function

We do not usually refer to the GUI *event* itself. Instead, we tend to refer to the actions that are associated with the GUI *events*.

5.1.1 Actions

An *action* is a named expression. *Actions* are normally associated with events. For example, when an event happens, the *action* attached to the event is identified and the associated expression is evaluated. *Actions* can be defined in any of the components of a report, but they are generally invoked from layouts.

Actions can be divided into four categories: Mouse Gesture, Menu Gesture, Window Gesture, and Shortcut. Below, each category is listed with its respective actions.

MOUSE GESTURE ACTIONS (for a right-handed mouse)

- select: single-click with the left mouse button
- dblclick: double-click with the left mouse button
- drag: press and hold with shift left mouse button (middle button on a three-button mouse)
- abandon: type the <Esc> key
- confirm: type the <Return> or <Enter> key

MENU GESTURE ACTIONS

These actions are all invoked by choosing the appropriate menu item or toolbar button.

- add_cell
- add_col
- add_row
- create_model_tag
- delete_model_tag
- display_model_tag
- left_right
- move - identifies when you request to balance the load in the bucket with the available capacity
- move_in - identifies when you request to balance the load in the bucket with the available capacity
- move_in_or_off - identifies when you request to balance the load in the bucket with the available capacity
- move_in_out - identifies when you request to balance the load in the bucket with the available capacity
- move_out - identifies when you request to balance the load in the bucket with the available capacity
- move_out_or_off - identifies when you request to balance the load in the bucket with the available capacity
- swap_down - identifies when you want to manually sort by moving the cell with focus one row down
- undo_to_mark
- update - identifies when you request that the report be refreshed with the latest information in the engine

WINDOW GESTURE ACTIONS

These actions are invoked through normal window navigation (e.g. picking a new tab).

- on_layout_hide - identifies when a layout changes from being visible to invisible
- on_layout_show - identifies when a layout changes from being invisible to visible
- on_report_open
- on_report_close
- on_report_raise

SHORTCUT ACTIONS

These actions provide pre-determined buttons that should invoke the indicated behavior when pressed.

- report: provides a blue "drill down" icon that should display another report that is related in some way to the data shown in the cell
- choose: provides a "choose" icon that should offer a choice of values for this cell. This is usually done by dispatching the system-defined *model_choose* action.
- more: provides a "more" icon that should display further information about the cell value
- map: provides a "map chart" icon that should display the map chart for the cell value

5.1.1.1 Action Syntax

To declare an action, the syntax is as follows:

```
action: action_name = expression;
```

where *action* must be typed as shown, and substitutions must be made for *action_name* and *expression*. The *action_name* can be anything. Although using one of the pre-defined names is permitted, it will cause that action to be re-defined. The *expression* can be any valid OIL expression. Any action's *expression* can refer to the following:

- all model fields
- built-in functions
- data type conversions
- cell pointer format
- any enum constant (e.g. LINK)
- numbers
- "quoted" strings

For actions defined in report files, expression can contain any of the following values:

- parameters that were passed to this report
- any variables and computes defined in the report

For actions defined in layout files, expression can contain any of the following values:

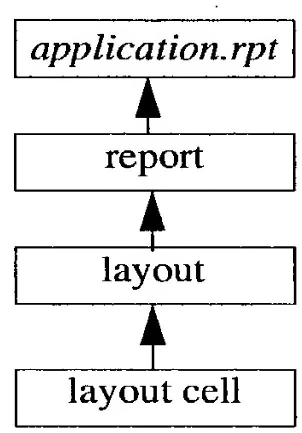
- parameters that were passed to the associated worksheet
- any variables and computes defined in the worksheet
- the worksheet's cell names (e.g. C14)

Actions can be declared within layout cells (as layout properties), or as report properties. System-defined actions are specified in *application.rpt*.

5.1.1.2 Action Lookup

When a user triggers a GUI event, OIL will associate an action with that event, and then proceed to search for a usable action declaration. RHYTHM SCP finds an action declaration to use by searching for definitions in a bottom-up order, as illustrated in FIGURE 24.

FIGURE 24 Action Lookup



5.1.1.3 Invoking Actions

Most layout cells may have one or more actions specified for invocation. For example,

```
[ Z3 = action: select = echo("hello"); ].
```

declares that if cell Z3 is *selected*, then the OIL expression `echo("hello")` will be evaluated.

In addition, actions can be directly invoked from OIL expressions by *dispatching*. The `dispatch` function is useful for redirecting a mouse gesture to another action. For example,

```
action: choose = dispatch("model_choose");
```

declares that if the *choose* action is recognized, then OIL should look up the *model_choose* action and evaluate its associated OIL expression.

5.1.1.3.1 Example Action

The following worksheet and layout files illustrate the use of actions.

supply_chain_model.wrk

```
normal supply_chain_model
(Supply_Chain sc)
{
[ C1 name = sc.name; ]
[ E1 plan = sc.plans.first; ]
.....
}
```

supply_chain_model.lyt

```
spreadsheet supply_chain_model
{
worksheet: supply_chain_model
[ C1=control: button; action: select = echo(name); ]
[ E1=action: map = display_report("plan_map", E1);
action: report = display_report("plan_editor", E1); ]
}
```

Layout cell C1 will be rendered as a button; if pressed, it will echo the name of the supply_chain. Layout cell E1 has two short-cut actions: map will display a map chart report, report will display a plan editor. Clicking on the appropriate icon at E1 will display the corresponding report.

5.1.1.4 Commonly Associated Actions and Controls

There are actions which are *normally* associated with each control. Therefore, the report writer should have these in mind when designing a report. However, there are also actions that would be *reasonable* to attach to a control (would not surprise the user), but are not typically associated together.

There are also cases where nothing is appropriate, or where one or more actions should be specified by the control's parameters instead of via a separate action definition. For example, a *tool_button* specifies its own *select* action. Table 9 lists all controls and actions that would normally be associated with them, and the actions that could reasonably be associated with them.

Table 9: Associated Actions and Controls

Control	Normal Action	Reasonable Actions
bar	select, menu	*
button	select	*
checkbox	*	confirm (*OR* select, not both)
combo	*	confirm (*OR* select, not both)
combo_popdown	*	confirm (*OR* select, not both)
filler_bar		
gantt_axis	NO ACTIONS SUPPORTED	
gantt_bar	select, menu	*
general	*	menu, report, more, choose, map, select, (confirm if cell is editable)
indented_general	*	menu, report, more, choose, map
label	*	*
layout	NO ACTIONS SUPPORTED	
line	select, menu	*
line_rate	select, menu	*
line_step	select, menu	*
list_bar	select, menu	*
map_connect	select, menu	*
map_node	select, menu	*
menu_item	ACTION IS A PARAMETER OF THIS CONTROL	
menu_radio_item	ACTION IS A PARAMETER OF THIS CONTROL	
outline_general	*	menu, report, more, choose, map

Table 9: Associated Actions and Controls

Control	Normal Action	Reasonable Actions
percentage_bar	select	*
radio_button	select	confirm
separator	NO ACTIONS SUPPORTED	
slider	*	confirm (*OR* select, not both)
spinner	*	confirm (*OR* select, not both)
submenu	NO ACTIONS SUPPORTED	
time_buckets	NO ACTIONS SUPPORTED	
tool_button	ACTION IS A PARAMETER OF THIS CONTROL	

5.1.2 Bindings

Bindings map keyboard events to defined actions. For example, a keyboard accelerator is a *binding*.

Binding syntax is as follows:

```
bind: keyboard combination = action;
```

For example,

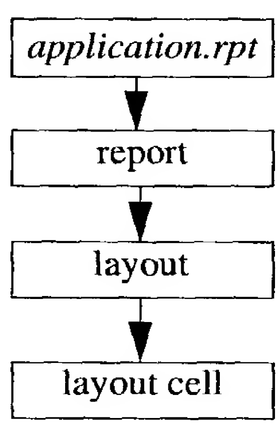
```
application() {
  action: close = close_this_report();
  bind: command+c = close;  // command + means "control"
}
```

In the example above, the binding *command+c* will cause the action *close* to be invoked when the user types <Ctrl><C>. The action *close* will then invoke the OIL function *close_this_report*.

5.1.2.1 Binding Lookup

RHYTHM SCP knows which *binding* definition to use according to the *binding* lookup. OIL searches for definitions in a top-down order, as illustrated in FIGURE 25.

FIGURE 25 Binding Lookup



Thus, a locally defined keyboard accelerator will never override one that is defined in *application.rpt*.

Section 6 Using Variables and Computes

You are now going to learn how to apply *variables* and *computes* to the reports you have been learning how to build. Through the use of these declarations, you can add dimension to certain controls, such as *combo_popdowns* and *radio_buttons.*

6.1 Worksheet Evaluation

A worksheet cell's *set* expressions are evaluated only when the cell is edited (e.g. typing, clicking a checkbox, etc.). A worksheet cell's *get* expressions are re-evaluated when one of the following occurs:

- the worksheet is first created and initialized
- an input parameter changes value
- any cell's set expression is evaluated

Efficiency hint: Many times the worksheets in a report share information. Try to put the shared information in the report and pass it to all the worksheets that need it, instead of re-computing the same things in each worksheet.

6.1.1 Variable and Compute Declarations - Overview

Variables and *computes* are mechanisms for holding values without requiring a worksheet cell. Unlike cells, they are usually used as intermediates and are often not visible within the UI. Unlike normal worksheet cells, they can hold values of the type List. *Variables* and *computes* are similar to parameters in that they are initialized when the worksheet/ report is instantiated.

Syntax for *variable* and *compute* declarations is straightforward, as shown below:

```
variable variable_name = expression;

compute compute_name = expression;
```

These declarations can be made in worksheets or in reports. Just as layouts have no parameters of their own, layouts do not declare their own

variables or *computes*. They can, however, access those of the corresponding worksheet.

6.1.1.1 Variables

Variables maintain the initial value, even if the initializing expression's value changes subsequently. *Variables* can only change via the *set_variable* OIL function. For example, the syntax of *set_variable* is:

```
set_variable (variable_name, expression)
```

The type of the *variable* is determined by the type of the value returned by the initializing expression. The type of the value assigned to a *variable* via *set_variable* must match the type of the *variable*. For instance, if the initial *variable* returned a Number, the new *variable* value must also be an Number. (OIL will make automatic type conversions where possible and appropriate, e.g., from Integer to Number.)

6.1.1.1.1 Using Variables

Variables can be used for the following:

- Initializing parameters for component layouts
- Transferring information between worksheets

For example, a report can use *variables* as shown below:

```
resource_plan_editor (Resource_Plan rp) {
  variable my_plan = rp.owner.owner;
  layout: L1 (my_plan);     // uses a worksheet "W1"
  layout: L2 (my_plan); } // uses a worksheet
                                              "W2"
```

The worksheets *W1* and *W2* use the same Plan Model, the one held in the report variable *my_plan*, as their incoming parameter. If one worksheet makes a change to *my_plan*, **both** see the change.

Another example of using *variables* is shown below. This example illustrates a cell that is initialized before allowing users to change its value.

```
normal bb (Buffer_Plan bp) {
  variable dd = bp.owner.owner.horizon.start;
                    // initial value of F2
  .....
  [ E2 = "Enter Dates"; ];
  [ F2 = dd; ];
  .....
}
```

Note: The *set* expression for *F2* is `set_variable`, i.e. OIL treats `[ F2 = dd; ];` as `[ F2 = dd, set_variable (dd,#); ];`. This is because OIL will try to create a *set* expression by using a *set_* function on the *get*'s outermost function, if the *set_* function exists. By definition, the *set_* function for a variable is *set_variable.*

Therefore, when the contents of F2 are, for instance, interactively edited, what occurs is that the inputs (which are represented by *#*) are used to set the value of *dd*. This in turn changes the value of F2.

6.1.1.2 Computes

Compute updates its value whenever the initializing expression's value changes. This happens whenever the worksheet/report is re-evaluated. Unlike variables, the *set_variable* expression cannot be used with *compute*. However, *computes* are similar to variables in that the type of the *compute* is determined by the type of the value returned by the expression.

6.1.1.2.1 Using Computes

A *compute* can be used to represent a complex computation. The *compute* item can then be used whenever the result of that expression is needed. This will enable RHYTHM SCP to perform more efficiently. Since a compute is not a cell, its value can easily be hidden from the end-user.

An example of using a *compute* is shown below:

```
normal siteplanwork (Site_Plan sp) {
  compute x = ...//some really messy expression
  that returns a List that you'd rather not
  calculate more often than absolutely
  necessary;
  [ E2 = x.first; ];
  [ E3 = x.element (2); ];
  [ E4 = x.element (3); ];
  .....
}
```

Note: Cells *E2*, *E3*, and *E4* are read-only cells because there is no *set_list_element* function that OIL can use to automatically create a set expression.

6.1.1.2.2 Putting It All Together

Given the following OIL expression in a worksheet:

```
variable var = supply_chains.first.
  sites.first;
variable other = supply_chains.element(2).
  sites.first;
compute comp = var.name;
[ A1 = comp, do(define(tmp, var),
  set_variable(var, other),
  set_variable(other, tmp)); ]
        // set_expression swaps var and other
```

assume the layout uses the default control, *general*, for cell *A1*; therefore, *A1* displays as an editable text cell. Assume also that *var* = *"love"*, *other* = *"hope"*; therefore *comp* = *"love"* and *A1* contains *"love"*.

Now, think through this as a computer would. The user types in cell *A1* and presses the <Return> key. The <Return> triggers the *confirm* action found in application.rpt, which calls the *edit_cell* function, which in turn invokes the set expression for the cell. *var* and *other* exchange values (note that the user's typed input is ignored since # is not used in the *set* expression), and the worksheet is re-evaluated, which means the following:

- *comp* is re-initialized
- the get expression for cell *A1* is evaluated

At the end, *var* = *"hope"*, *other* = *"love"*, *comp* = *"hope"*, and *A1* displays *"hope"*.

6.1.1.3 make_type

make_type is an OIL function that returns a value with a specified type. *make_type* requires a specific syntax which is shown below:

```
make_type (value, Type_Name)
```

where *make_type* must be typed as shown and substitutions must be made for *value* and *Type_Name*. In the example above, *make_type* returns a value with the type *Type_Name*.

Here is an example of using *make_type* for a cell that needs a List whose contents are not yet known:

```
normal bb (Buffer_Plan bp) {
variable dd = make_type(nonexistent, List[Site]);
...
  [ F2 = dd.first;];    // won't display until the List
                // has at least one member, via
                // set_variable(dd, ...);
....
}
```

Here is another example of using *make_type*:

```
compute x = make_type ("False", Logical);
```

will initialize *x* to the logical value of *false*, whereas:

```
compute x = "False";
```

initializes *x* to the string value *false*.

6.1.2 Re-using Definitions

Definitions to be used more than once may be collected into files and included in reports or worksheets using the *#include* declaration:

```
#include "filename";
```

Example applications of re-using definitions are listed below:

- Common variable definitions (in *.var* files)
- Common action definitions (in *.act* files)
- Standard menubar and toolbar definitions (usually in *.mnu* files)

An example of using the *#include* declaration is shown below:

```
plan_editor (Resource_Plan rp) {
....
  #include "planning.mnu";
  #include "problems_filter.var";
  #include "planning.act";
```

where the Resource Plan Editor includes the *planning.mnu*, *problems_filter.var*, and *planning.act* files.

Note: The *#include <filename>* expression acts the same as if each declaration in the file were simply typed into the report. In the example above, *#include planning.mnu* acts the same as if each declaration in *planning.mnu* had been typed into the *plan_editor* report. Therefore, if the contents of *planning.mnu* are subsequently altered and `reload_report_definitions` is invoked, the *plan_editor* report will not see the changes until the *plan_editor* is also re-initialized.

Section 7

Using Replicating Worksheets

Up to this point, you have been learning about writing reports using normal worksheets. Now, you are going to begin learning to use replicating worksheets. An important use of replicating worksheets is in the axis cross layout. This section will familiarize you with using replication and axis cross.

7.1 Replicating Worksheets

In a replicating worksheet, one or more cells in the worksheet evaluates to a list of values. The expression associated with the replicating cell operates on each element in the list of values. Conceptually, a replicating worksheet is like having a normal worksheet for every value of the replicating cell. For example,

```
replicating sc (Supply_Chain supply_chain) {
  [ A1 site = supply_chain.sites; ]
  [ A2 managed = site.managed; ]
}
```

where *site* is an alias for cell A1 and *managed* is an alias for cell A2.

Here, A1 will have as many replicates as there are elements in the list supply_chain.sites. A2 will have as many replicates as A1 has; each of A2's replicates calculates the managed field of one of the sites in A1.

The number of replicates in a cell can potentially be quite larger. To see this, consider the following example worksheet:

```
replicating stuff (Plan p) {
[ A1 site_plan = p.site_plans; ]
[ A2 request = site_plan.requests; ]
[ A3 dr = request.delivery_requests; ]
}
```

Suppose that the incoming parameter "p" has three site_plans, all of which are LINK. Suppose that one has five requests, and the other two have severn requests each. Finally, suppose that all requests happen to have two delivery_requests. Then the number of replicates in each cell is:

```
Cell      Replicates
A1        1 + 1 + 1 = 3
A2        1 * 5 + 1 * 7 + 1 * 7 = 19
A3        1 * 5 * 2 + 1 * 7 * 2 + 1 * 7 * 2 = 38
```

If, on the other hand, each site_plan has ten requests, then A3 would have sixty replicates. If we add a cell, A4, as below:

```
[ A4 ir = dr.item_requests; ]
```

and there are three item_requests for each delivery_request, then A4 would have one hundred fourteen replicates. Clearly, as one traverses through the model hierarchy using replication it is possible to have cells with hundreds or thousands of replicates.

Cells in replicating worksheets may be assigned a *title* specification. This is a convenient way to give a title to the list of values in the cell for use by layouts that support titles (e.g. axis cross). The example below illustrates the use of the *title* specification:

```
replicating sc (Supply_Chain supply_chain) {
  [ A1 site = supply_chain.sites; ]
  [ A1.title = "Site"; ]
  [ A2 managed = site.managed; ]
  [ A2.title = "Managed"; ]
}
```


7.1.1 Types of Replication

There are different types of replications which are supported by a replicating cell depending on the relationships between cells in the same worksheet:

- Independent
- Single Dependent
- Multiple Dependent

Note that cyclical dependencies (e.g. cell A1's value depends on cell A2's value and cell A2's value depends on A1) are not allowed in replicating worksheets. Dependency graphs are useful in debugging cycles and other replication errors.

7.1.1.1 Independent Replication

In an independent replication, the cell expression generates a list of values and is independent of any other replicating cell in the worksheet. For example,

```
replicating sc (Supply_Chain supply_chain) {
  [ A1 site = supply_chain.sites; ]
}
// A1 is independent
```

The dependency graph for this example would look like this:

A1

7.1.1.2 Single Dependent Replication

In a single dependent replication, the cell expression generates a value that depends on one other cell in the worksheet. Either of the cells can replicate. For example,

```
replicating sc (Supply_Chain supply_chain) {
  [ A1 site = supply_chain.sites; ]
  [ A2 name = site.name; ]
}
// A2 is single dependent on A1
```

The dependency graph for this example would look like this:

```
A1
|
A2
```

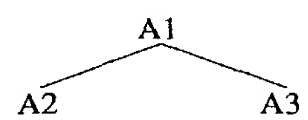
Any cell that appears in a *get* expression establishes a dependency, even if its value is ultimately not used to calculate the cell's value. In this example, A1 is appearing in the *get* expression for A2, thereby creating the dependency. The dependency is there, even if we make the following nonsensical change:

```
[ A2 name = if(A1 == A1, "Hello", site.name); ]
```


As another example of single dependent replication, consider the following:

```
replicating sc (Supply_Chain supply_chain) {
  [ A1 site = supply_chain.sites; ]
  [ A2 name = site.name; ]
  [ A3 opn = site.operations; ]
}
```

The dependency graph in this case would look as follows:



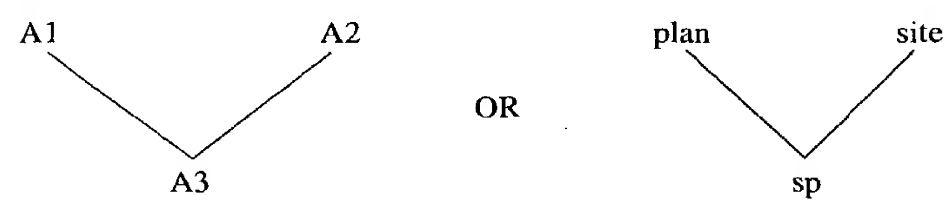
In this example, A2 and A3 are dependent on A1. A3 is also a *nested* replication, because it generates a list for every replicated value of A1.

7.1.1.3 Multiple Dependent Replication

In a multiple dependent replication, the cell expression generates a value that depends on more than one cell in the worksheet. For example,

```
replicating site (Supply_Chain sc, Site x) {
[ A1 plan = sc.plans; ]
[ A2 site = x; ]
[ A3 sp = plan.site_plans.filter(#.site != x); ]
}
```

The dependency graph for this example would look like the following:



where A3 is dependent on A1 and A2. The value of A3 cannot be generated without first generating the values for A1 and A2.

7.1.2 List Functions

"List" functions create and manipulate lists of values. There are several "list" functions:

- count
- filter
- for_each
- sort
- sort_stable
- sublist
- unique
- contains
- found
- element
- find
- find_or_nonexistent
- find_or_create
- key
- multi_keyed_list
- key_names
- multi_key_bucketize
- bucket_list_by_date
- current_depth
- list_index
- first
- last
- list
- bucketize
- bucket_list
- bucket_symbols
- keyed_list
- keyed_element
- keys
- recurse
- recurse_and_trim
- integers
- multi_key
- multi_keyed_element
- key_values
- multi_key_bucketize_element
- bucket_list_by_key

"List" manipulation functions such as *for_each* iterate over each element of the list. Other functions, such as *do* or *define*, are useful when combined with "list" functions. **Syntax Note:** The element in the list that is being operated on is represented by #.

7.1.2.1 count Function

count determines the number of elements in a list. The list can be of any type. In the expression `count(list(val1, val2, val3))`, *count* returns 3 as the number of elements in the list. FIGURE 26 shows examples of the count function:

FIGURE 26 count Function Example

```
count(list(5, 10, 15))      //returns 3
list("a", "b", "c", "d").count// returns 4
```

7.1.2.2 list Function

list creates a list of values. Each value in the list must have the same *type*. If any of the parameters is a list, the elements of that list are included directly into the list. The expression `list(val1, val2, ..., valn)` makes a list containing the values *val1, val2, ..., valn*. FIGURE 27 shows an example of the list function within an expression:

FIGURE 27 list Function Example

```
list(3, list(2, 4))
```

The example above would create a list containing the values *3, 2, 4* (in that order).

7.1.2.3 sublist Function

sublist returns a portion of another list as specified by the two integer parameters. In the expression `sublist(List[Void], Integer1, Integer2)`, *Integer1* is the position of the starting element and *Integer2* is the number of elements. FIGURE 28 illustrates the use of *sublist*:

FIGURE 28 sublist Function Example

```
list(1, 2, 3, 4, 5).sublist(2, 2)   returns (2, 3)
list(1, 2, 3, 4, 5).sublist(5, 2)   returns (5)
list(1, 2, 3, 4, 5).sublist(6, 10)  returns ()
list(1, 2, 3, 4, 5).sublist(0, 3)   error
```

7.1.2.4 for_each Function

for_each traverses a list applying *Expression* to each element and placing the result in a new list. The expression `for_each (List[Void], Expression)` exhibits proper syntax for the *for_each* function. FIGURE 29 shows an example of using *for_each*:

FIGURE 29 for_each Function Example

```
list(20, 6).for_each(# + 3)     returns (23, 9)
```


7.1.2.5 filter Function

filter traverses a list applying the logical *Expression* to each element. If the result is *true*, then the element is added to a new list. The expression `filter (List[Void], Expression, Integer)` exhibits proper syntax for the *filter* function. The Integer parameter is an optional count for the maximum size for the result (defaults to infinite). This parameter provides a limit which stops the search through the list once it has found the specified number of elements. FIGURE 30 displays an example of the *filter* function:

FIGURE 30 filter Function Example

```
list(20, 6).filter(# == 3)  returns (20)
```

7.1.2.6 recurse Function

The *recurse* function creates a result list from the input list of values, then evaluates *Expression* on each member, adding the results of the evaluation to the result list. *Expression* is an OIL expression that returns a list. The resulting list maintains depth information for the elements in the list, which can be used to intelligently indent displayed results. For example, starting with a single-level Bill of Materials, *recurse* can calculate the full BOM and preserve the depth information needed to show the indented BOM. FIGURE 31 shows an example of using the *recurse* function.

FIGURE 31 recurse Function Example

```
list(20, 6).recurse(#.my_factors)
```

In FIGURE 31, *my_factors* (which is an integer) returns a list of the factors of its input integer. The result starts as (20, 6). Then 20 is passed to *my_factors*, which returns (2, 4, 5, 10). The result is appended after 20: (20, 2, 4, 5, 10, 6). The list "knows" that 2, 4, 5, and 10 are "underneath" 20. Next, 2 is processed, etc. The final result is (20, 2, 4, 2, 2, 5, 10, 2, 5, 6, 2, 3).

Note: The depth information is lost if the list generated by *recurse* is then operated upon by another list function such as *filter*.

FIGURE 32 illustrates the correct way and the incorrect way to use the *recurse* function:

FIGURE 32 Correct and Incorrect Use of recurse

```
replicating product_groups (List[Product_Group]
product_groups) {
       [ A1 right = product_groups.recurse(#.sub_groups
           .for_each(#.product)group)); ]
       [ A2 wrong = product_groups.recurse(#.sub_groups)
           .for_each(#.product_group); ]
}
```

In the example above, cell *A1* stores a hierarchical list of all the product groups under the input parameter list of product groups. Therefore, the depth information is preserved in *A1* because the *for_each* is performed while *recurse* is still being calculated. In contrast, *A2* passes the result of *recurse* to *for_each*, which cannot preserve this information.

7.1.2.7 recurse_and_trim Function

recurse_and_trim behaves identically to the *recurse* function except that it has an additional boolean filter expression which is used to trim the resulting tree. The filter expression argument is a little different from the normal *filter* function because an element for which the expression returns *false* will still be included if one of its children's filter expressions returned *true*. In other words, this filter is specifically designed to trim dead branches of the tree.

The correct syntax for the *recurse_and_trim* function is `recurse_and_trim (List[Void], Expression, Expression).`

7.1.2.8 sort Function

The *sort* function orders a list based on the sorting expression. The expression `sort (List[Void], Expression)` exhibits proper syntax for the *sort* function. *sort* processes a list until the sorting expression is *true* for all adjacent elements. (**Note:** Use 'a' and 'b' as element place holders in the expression.) FIGURE 33 shows an example of using the *sort* function:

FIGURE 33 sort Function Example

```
list(4, 6, 8, 2).sort(a < b);
  returns list(2, 4, 6, 8)
```

7.1.2.9 sort_stable Function

sort_stable behaves identically to *sort*, except that *sort_stable* ensures that the ordering of the equal elements in the input list is maintained in the output list. *sort_stable* orders a list based on an expression. In contrast, *sort* processes a list until the sorting expression is *true* for all adjacent elements. The expression `sort_stable ( List[Void], Expression)` illustrates the correct syntax for the *sort_stable* function. (**Note:** Use 'a' and 'b' as element place holders in the expression.)

sort_stable is much slower than *sort*; therefore, use *sort* whenever possible. FIGURE 34 shows several examples of using *sort_stable*:

FIGURE 34 sort_stable Function Example

```
list(4, 6, 8, 2).sort_stable(a<b);//returns list(2, 4, 6, 8)
list("t", "h", "m").sort_stable(a>b)     //returns list("t", "m", "h")

list(25, 26, 17, 18, 19).sort_stable(integer(a/10) < integer(b/10));
//This sorts by the most significant digit, and returns
//list(17, 18, 19, 25, 26)

@~list(25, 26, 17, 18, 19).sort(integer(a/10) < integer(b/10));
//This sorts by the most significant digit, and returns
//list(19, 17, 18, 26, 25)
            Notice that least-significant digits are in a different order.
```

The following expressions are provided to illustrate the difference between *sort_stable* and *sort*:

```
operations.sort(a.name < b.name).sort_stable
   (a.category < b.category);
```

The expression above sorts operations using *category* as the primary key, and *name* as the secondary. The following does the same thing but is over twice as fast:

```
operations.sort(if (a.name != b.name, a.name <
    b.name, a.category < b.category));
```


7.1.2.10 unique Function

unique removes duplicate elements from a list, thereby returning a list of unique elements. The expression `unique( List[Void])` shows the correct syntax for *unique*. FIGURE 35 shows an example of using *unique* within an expression:

FIGURE 35 unique Function Example

```
unique(list(1, 2, 3, 2, 1));    // returns list(1, 2, 3)
```

7.1.2.11 contains Function

contains determines if a list contains a certain element. When given a list of models, *contains* searches using the Model's key field, but only returns TRUE if the matching key field model is the exact model instance given. FIGURE 36 illustrates the use of the *contains* function:

FIGURE 36 contains Function Example

```
supply_chains.contains("test");//returns TRUE if there is a supply chain
        //named "test in the list of supply chains

list(1, 2, 3, 4).contains(0);//returns FALSE
contains(list(3, 5, 7, 9), 7);// returns True
```

7.1.2.12 found Function

found behaves identically to *contains* with the exception of its usage on models. *found* loosens the match criteria to merely key_fields (just like *find*, only *found* returns a Logical instead of the element), whereas *contains* matches the key field of the exact model instance. FIGURE 37 below shows examples of the *found* function:

FIGURE 37 found Function Example

```
supply_chains.found("test");//returns TRUE if there is a supply chain
              //named "test" in the list of supply chains

list(1, 2, 3, 4).found(0);//returns FALSE
found(list(3, 5, 6, 9), 7);//returns TRUE
```

7.1.2.13 element Function

element returns the requested element from a list. Passing 1 in as the second argument returns the first element in the list. The expression `element(List[Void], Integer)` illustrates the proper syntax for the *element* function. FIGURE 38 shows examples of using the *element* function:

FIGURE 38 element Function Example

```
element(list(5, 10, 15), 1);     //returns 5

list("a", "b", "c".element(2);   //returns "b"
list(1, 2, 3, 4).element(5);     //returns an error
```


7.1.2.14 find Function

find searches through a list of models and returns the first model whose key field matches the given value. This version of *find* is used when the first argument is a list of models. An error is returned if no match is found. The correct syntax for the *find* function is as follows:

```
find (List[Void],Void)
```

Examples of using the *find* function are shown in FIGURE 39:

FIGURE 39 find Function Example

```
supply_chains.find("test");//returns the supply chain named "test"

sites.find("Dallas");//returns the site named "Dallas"
```

7.1.2.15 find_or_nonexistent Function

find_or_nonexistent behaves identically to *find* by searching a list of models and returning the first model whose key field matches the given value. However, *find_or_nonexistent* returns `nonexistent` when no match is found. *find_or_nonexistent* is also used when the first argument is a list of models.

FIGURE 40 shows two examples of the *find_or_nonexistent* function:

FIGURE 40 find_or_nonexistent Function Example

```
supply_chains.find_or_nonexistent("test");
                //returns nonexistent

site.find_or_nonexistent("Dallas");
                //returns the site named "Dallas"
```

7.1.2.16 find_or_create Function

find_or_create searches through a list of models and does one of the following:

- returns the first model whose key field matches a value
- if the model does not exist, creates and appends it to the end of the list followed by returning the newly created model

find_or_create is only available to a list of models.

FIGURE 41 provides an example to differentiate between the *find_or_create* function and the *find* function. In this example, assume that the supply chain named "test" does not exist. Observe that the first expression yields an error because it did not find "test." The other expressions return the model named "test."

FIGURE 41 find_or_create Function Example

```
supply_chains.find("test");//returns an error
supply_chains.find_or_create("test");
                              //creates and returns the model named "test
supply_chains.find("test");//returns the model
                              //named "test"
supply_chains.find_or_create("test");
                              //returns the model named "test"
```


7.1.2.17 list_index Function

list_index searches through a list of models and returns the position in the list of the first model whose key field matches the given value. It also searches an arbitrary list for a match (using the == operator). *list_index* reports an error if the list element is not found. FIGURE 42 shows an example of using *list_index* within an expression. It also provides an example illustrating the difference between *list_index* and *find*:

FIGURE 42 list_index Function Example

```
list("a", "b", "c").list_index("b");//returns 2

some_list(some_list.list_index(key));

                          is equivalent to...

                                   some_list.find(key);
```

7.1.2.18 first Function

first searches through a list (of any type) and returns the first element within the list. If the list is empty, *first* returns `nonexistent`. The expression `first (List[Void])` exhibits the correct syntax for the *first* function. FIGURE 43 shows examples that illustrate the proper usage of the *first* function:

FIGURE 43 first Function Example

```
first(list(5, 10, 15));      //returns 5
list("one", "two", "three").first;//returns "one"
```

7.1.2.19 last Function

last searches through a list (of any type) and returns the last element of the list. *last* returns `nonexistent` if the list is empty. FIGURE 44 shows two examples of using the *last* function:

FIGURE 44 last Function Example

```
last(list(5, 10, 15));          //returns 15
list("one", "two", "three").last;//returns "three"
```

7.1.2.20 bucketize Function

bucketize searches through a list and returns a list of values that are organized by quick retrieval characteristics. Use *bucketize* when you want to be able to quickly retrieve values from a large list that is date-based in nature. The correct syntax for *bucketize* is shown below:

```
bucketize (List[Void], List[Date_Range],
  Expression, Expression)
```

bucketize organizes the input list into a table with one column per date_range (using the second parameter), and any number of rows (indexed by a symbol). The intersection (table entries) is a list which you can access using the *bucket_list* function. The last two parameters are used to determine which of the input items go into which symbol/ date_range intersection list.

The following is a description of the workings of *bucketize*:

1. First, *bucketize* takes the list of date_ranges, and makes a bucket for each one (Note that in a list of date_range buckets, any two adjacent buckets, A and B, must have: A.end == B.start)
2. Then it loops over the list of values and does the following for each:
   • evaluates the date expression
   • finds the bucket in which that date expression fits (i.e. bucket.start <= date < bucket.end). (**Note:** this implies that given a bucket where bucket.start == bucket.end, nothing can ever fall within it)
   • evaluates the symbol expression
   • puts this value indexed by this symbol to the list in this bucket

7.1.2.20.1 Example Usage of bucketize

Consider a list of item_promises. Put each one in the bucket whose date_range contains the given date as evaluated in the expression `#.owner.due.start`, and hash-keys it with the given Symbol as evaluated in the expression `#.item.name`.

```
variable ips = delivery_promises.for_each
(#.item_promises);
variable bl = bucketize(date_range_list,
#.owner.due.start, #.item.name);
```

The following is an extended user example. It is a contrived, non-model example, but it shows in painstaking detail the flow of data:

```
define(date_range_list, list(date_range
("95/01/01 00:00 / 95/01/02 00:00"),
   // bucket#1
date_range("95/01/02 00:00 / 95/01/06
00:00"), // bucket#2
date_range("95/01/06 00:00 / 95/01/10
00:00"), // bucket#3
date_range("95/01/10 00:00 / 95/01/14
00:00"), // bucket#4
date_range("95/01/14 00:00 / 95/01/20
00:00"))) // bucket#5
define(b_example, integers(5,10).
bucketize(date_range_list, date("95/01/00
   00:00") + time(#.string & " day"),
   #.string))
```

For the example above, we generate a date for each of the integer numbers between 5 and 10 (specifically: “95/01/05 00:00”, “95/01/06 00:00”, “95/01/07 00:00”, “95/01/08 00:00”, “95/01/09 00:00”, “"95/01/10 00:00”).

Therefore, integer 5’s value gets stored in bucket#2 since: bucket#2.start <= (5’s date) < bucket#2.end “95/01/02 00:00” <= “95/01/05 00:00” < “95/01/06 00:00”. Integer 6 ends up in bucket#3, (NOT in bucket#2, due to the . Integers 7, 8 and 9 also get put in the list in bucket#3 as well. Integer 10 ends up in bucket#4 (NOT in bucket#3, same reason as integer 6).

7.1.2.21 bucket_list Function

bucket_list is used to quickly retrieve a specific value (or values) from a list created by *bucketize*. *bucket_list* finds the bucket whose given date falls within its date_range (i.e. bucket.start <= date < bucket.end). It then retrieves any values that were stored in that bucket with a lookup key that matches the given symbol. FIGURE 45 shows examples of using *bucket_list*:

FIGURE 45 bucket_list Function Example

```
b_example.bucket_list(date("95/01/10 00:00"), "9") --> nothing
b_example.bucket_list(date("95/01/02 20:00"), "9") --> nothing
```

The following examples of *bucket_list* show how the given lookup date does not have to match the original date used to put the data in that specific bucket:

```
b_example.bucket_list(date("95/01/02 20:00"),
    "5") --> 5 b_example.bucket_list(date
    ("95/01/07 00:00"), "8") --> 8
b_example.bucket_list(date("95/01/06 00:00"),
    "8") --> 8 b_example.bucket_list(date
    ("95/01/06 00:00"), "9") --> 9
b_example.bucket_list(date("95/01/06 00:00"),
    "adam") --> nothing
b_example.bucket_list(date("95/01/09 23:59"),
    "8") --> 8
```


7.1.2.22 bucket_symbols Function

bucket_symbols is used to get at the list of unique Symbols (keys) that were used in building the given list during *bucketize*. The expression `bucket_symbols (List[Void])` displays the correct syntax for the *bucket_symbols* function. FIGURE 46 provides an example of using *bucket_symbols*:

FIGURE 46 bucket_symbols Function Example

```
b_example.bucket_symbols() --> "5", "6", "7", "8", "9", "10"
```

7.1.2.23 keyed_list Function

keyed_list is used when you want to be able to quickly retrieve values from a large list. It loops over the list of values and does the following for each:

- Evaluates the symbol expression
- Stores this value, indexed by this symbol, in the new list

The proper syntax for *keyed_list* is shown below:

```
keyed_list (List[Void], Expression)
```

FIGURE 47 shows examples of the *keyed_list* function:

FIGURE 47 keyed_list Function Example

```
define(keyed, integers(1, 10).keyed_list(#.string & "0"));
```

This expression stores the integer value 1 through 10 with the respective key-symbo of "10", "20", "30", "40", "50", "60", "70", "80", "90", "100"

7.1.2.24 keyed_element Function

keyed_element is used to quickly retrieve a specific value (or values) from a list created by *keyed_list*. The proper syntax for the *keyed_element* function is shown below:

```
keyed_element (Symbol)
```

FIGURE 48 provides examples of using the *keyed_element* function:

FIGURE 48 keyed_element Function Example

```
keyed.keyed_element("50") --> 5
keyed.keyed_element("5")  --> nothing
```

7.1.2.25 keys Function

The *keys* function is used to retrieve the list of unique Symbols (keys) that were used in building the given list during *keyed_list*. The correct syntax for the *keys* function is `keys (List[Void])`.

FIGURE 49 shows an example of the *keys* function:

FIGURE 49 keys Function Example

```
keyed.keys()--> "10", "20", "30", "40", "50", "60", "70",
                "80", "90", "100"
```


7.1.2.26 integers Function

The *integers* function generates a list of integers from the minimum to maximum. This is used as a general looping mechanism when combined with the *for_each* function. The syntax for the *integers* function is `integers (Integer, Integer)`. The following are examples of how the *integers* function works:

```
integers(1, 5) -> integers(1, 5) = 1, 2, 3, 4, 5

integers(5, 4) -> integers(5, 4) = 4, 5

integers(5, 5) -> integers(5, 5) = 5

integers(5, 6) -> integers(5, 6) = 5, 6
```

FIGURE 50 shows an example of using *integers* within an expression:

FIGURE 50 integers Function Example

```
integers (1, 3).for_each(echo(#.string));    //returns 1, 3
```

7.1.2.27 key Function

The *key* function returns a key definition for use by the *multi_key* function. The syntax for the *key* function is `key (Symbol, Symbol, Logical, Logical)`.

FIGURE 51 shows an example of how the *key* function works:

FIGURE 51 key Function Example

```
define(key1, key("item", #.name, #.name != "leg", TRUE);
```

7.1.2.28 multi_key Function

The *multi_key* function returns a list of key definitions for use by the *multi_keyed_list* and *multi_key_bucketize* functions. The syntax for the *multi_key* function is `multi_key (Symbol, Void)`.

FIGURE 52 shows an example of using the *multi_key* function:

FIGURE 52 multi_key Function Example

```
define(mk, multi_key("Item", key1, key("where", #.owner.name, #.owner
            .name != "CANADA", TRUE)));
```

7.1.2.29 multi_keyed_list Function

The *multi_keyed_list* function returns a multiple-keyed list of lists of elements. The syntax for the *multi_keyed_list* function is `multi_keyed_list ( List[Void], List[Void], Expression)`.

FIGURE 53 shows an example of using the *multi_keyed_list* function:

FIGURE 53 multi_keyed_list Function Example

```
define(mkl, item_list.multi_keyed_list(mk, #.name, #.category,
#.delivery.name));
```


7.1.2.30 multi_keyed_element Function

The *multi_keyed_element* function returns a list of elements stored at the specified location(s). The syntax for the *multi_keyed_element* function is `multi_keyed_element (List{Void], Integer, Symbol, Symbol).`

FIGURE 54 shows an example of using the *multi_keyed_element* function:

FIGURE 54 multi_keyed_element Function Example

```
mkl.multi_keyed_element(2, "where", "USA");
```
Here we get the list of all the #.category that were stored for all site.names == "USA"

7.1.2.31 key_names Function

The *key_names* function returns the list of key names used to build a *multi_keyed_list* or *multi_key_bucketize* list. The syntax for the *key_names* function is `key_names (List[Void]).`

FIGURE 55 shows an example of using the *key_names* function.

FIGURE 55 key_names Function Example

```
mkl.key_names( ) = "item", "where"
```

7.1.2.32 key_values Function

The *key_values* function is used to retrieve the list of unique key values for the given name used to build a *multi_keyed_list* or *multi_key_bucketize* list. The syntax for the *key_values* function is `key_values (List[Void], Symbol).`

FIGURE 56 shows an example use of the *key_values* function:

FIGURE 56 key_values Function Example

```
mkl.key_values("item") = "table", "chair", "seat"
```

7.1.2.33 multi_key_bucketize Function

The *multi_key_bucketize* function returns a multiple-keyed and date_ranged (bucketized) list of lists of elements. The syntax for the *multi_key_bucketize* function is `multi_key_bucketize (List[Void], List[Void], List[Date_Range], Expression, Expression).`

For examples on using the *multi_key_bucketize* function refer to the *bucketize* and *multi_keyed_list* functions.

7.1.2.34 multi_key_bucketize_element Function

The *multi_key_bucketize_element* function returns a list of the element(s) stored at the specified location(s). The syntax for this function is `multi_keyed_bucketize_element (List[Void], Integer, Date, Symbol, Symbol).`

For examples on using the *multi_key_bucketize* function refer to the *bucketize* and *multi_keyed_list* functions.

7.1.2.35 bucket_list_by_date Function

The *bucket_list_by_date* function retrieves the whole list of values in the date_range bucket that the given date falls within. It returns a list of values organized by key (i.e. the same kind of list that comes results from *keyed_list*, and therefore usable by *keyed_element*, and *keys*). The syntax for this function is as follows:

```
bucket_list_by_date (List[Void], Date)
```

FIGURE 57 shows an example of using the *bucket_list_by_date* function:

FIGURE 57 bucket_list_by_date Function Example

```
b_example.bucket_list_by_date(date("95/01/06 00:00")) --> 6, 7, 8, 9
```

7.1.2.36 bucket_list_by_key Function

The *bucket_list_by_key* function retrieves the whole list of values in all the date_range buckets that match the given key. It returns a list of values organized by key (i.e. the same kind of list as comes out of *keyed_list*, and therefore usable by *keyed_element*, and *keys*). The syntax for this function is shown below:

```
bucket_list_by_key (List[Void], Symbol)
```

FIGURE 58 shows an example use of the *bucket_list_by_key* function:

FIGURE 58 bucket_list_by_key Function Example

```
b_example.bucket_list_by_key("8")
.keyed_element(date_range("95/01/06 00:00 / 95/01/10 00:00").string) --> 8
b_example.bucket_list_by_key("8")
.keyed_element("95/01/06 00:00 / 95/01/10 00:00")        --> NOTHING
```

7.1.2.37 current_depth Function

The *current_depth* function returns the current depth of a (recursively) replicating cell. The syntax for this function is `current_depth (Cell)`.

FIGURE 59 shows an example of using the *current_depth* function:

FIGURE 59 current_depth Function Example

```
current_depth(C1)
```


7.1.2.38 do, define, and do_file Functions

The *do* function (`do(expr1{, expr2,....})`) executes the expressions listed sequentially left to right. It returns the value of the last expression evaluated as the value of the *do*.

The *define* function (`define(temporary, expression)`) sets *temporary* to the value of the expression. *define* is used inside a *do* function. When writing a *define* function, treat *temporary* as a variable that is only visible in the containing *do* function.

A do_file is a collection of OIL expressions, separated by semicolons, that are evaluated sequentially. They are usually written with a *.in* or *.bat* extension. The *do_file* function (`do_file(filename)`) runs each expression in the file in text order. This function is commonly used in startup files. The *do_file* function is also used in debugging (returns the last thing it evaluates in the do_file statement). When used for debugging, the *do_file* ("-") function invokes an OIL "listener" in the window where the engine was invoked. However, this function works best when not using the *+stdout* option to redirect scp_engine messages to a text file (as we do in *runoil.cmd*).

```
Reading file \rhythm\OIL305\models\oilclass/product_groups.dat  using Product_Gr
oup_Group
Reading file \rhythm\OIL305\models\oilclass/product_groups.dat 13 records
Reading file \rhythm\OIL305\models\oilclass/forecast_entry.dat  using Forecast_E
ntry
Reading file \rhythm\OIL305\models\oilclass/forecast_entry.dat 10 records
Reading file \rhythm\OIL305\models\oilclass/bod.dat      using On_Hand
Reading file \rhythm\OIL305\models\oilclass/bod.dat 58 records
Handling requests from port 27111
Reading file /appl/reports/scp/domain.dat       using domain
Reading file /appl/reports/scp/domain.dat 119 records

Enter commands to be evaluated.  Type 'quit' or hit return to continue.
i2> 1+1
                1+1 = 2
i2> supply_chains.first
 supply_chains.first = i2 Furniture
i2> define(P,supply_chains.first.plans.first);
define(P,supply_chains.first.plans.first); = Active
i2> P
                  P = Active
i2> P.site_plans.for_each(#.plan_to_satisfy_all_requests);
P.site_plans.for_each(#.plan_to_satisfy_all_requests);
i2> quit
```

The "listener" reads, compiles, and executes the OIL expressions you have typed in. It then echoes the results. You can use *define* to create temporary values within the OIL "listener" which enables you to build complex calculations more easily. The figure above shows some expressions being processed by the OIL "listener". **Note:** In *.../application.rpt*, the <Shift><F12> key chord is bound to *do_file*("-"), so that from an OIL scp_ui, the "listener" can be interactively invoked by typing <Shift><F12>.

Note: Functional worksheets are the preferable method of creating OIL scripts (i.e. collections of OIL computations that can be invoked from batch clients). However, *do* and *do_file* can also be used for this purpose, for instance, in engine startup files.

BEFORE:

```
@if(!button,
   do(if(user.verbose,
     do(define(dash_line,
     "----------------------------------------" &
     "----------------------------------------"),
       echo(dash_line),
       echo(dash_line),
       echo("Starting Auto-Reschedule at: " & hour(now())
       .string & ":" & minute(now()).string & ":" &
       second(now()).string), echo(dash_line),
       echo(dash_line)), nonexistent), 1));
```

AFTER: (Create the file *verbose_report.fws* like this)

```
function verbose_report (String message)
{
[ A1 dash_line = "------------------------------------" &
   "-----------------------------"; ]
[ A2 blank_line = " "; ]
[ A3 = echo(blank_line); ]
[ A4 = echo(blank_line); ]
[ A5 = echo( "\t \t Engine Size: " & string(memory_size/
   1048576) & "Mega Bytes"); ]
[ A6 = echo(blank_line); ]
[ A7 = echo(blank_line); ]
[ A8 = echo(dash_line); ]
[ A9 = echo(message & " at: " & hour(now()).string & ":" &
   minute(now()).string & ":" & second(now()).string); ]
[ A10 = echo(dash_line); ]
[ A11 = echo(dash_line); ]
[ A12 return = 1; ]
}
```

And replaced the *do* above (BEFORE) with this:

```
if(!button,
   if(user.verbose,
     call("$I2_REPORTS/my_cust/verbose_report",
       "Starting Auto-Reschedule")));
```

7.1.2.38.1 Nested Scope in do and do_file

Variables (temporaries) in *do* and *do_file* functions have nested scope. Variables defined in one *do* can be accessed by another *do* nested with the first (e.g. `do(define(abc, 3), do(echo(abc + 2)));`). Variables defined in one *do_file* can be accessed by another *do_file* started from the first. For example,

```
do_file("FirstFile");
```

where the contents of the FirstFile are:

```
define(abc, 3);
do_file("SecondFile");
```

and the contents of the SecondFile are:

```
echo(abc + 2);
```

7.1.3 Axis Cross Layouts

The axis cross layout provides a tabular display of replicating cells. It can display a Cartesian cross-product of rows and columns. Since the layout is based on replicating worksheets, table rows or columns can have titles.

7.1.3.1 Definitions

The layout consists of the following elements:

- the X and Y axes - for values from replicating cells that determine the rows and columns
- the Cross - the product of values to be displayed across the X and Y axes

Cells in the rows or columns may be grouped to allow information to be displayed compactly.

7.1.3.2 Axis Cross Layout Syntax

Axis cross layouts require a specific syntax. This syntax is illustrated below:

```
axis_cross layout_name {
  worksheet: ws_name;
  X: row specification <CROSS>;
  Y: column specification <CROSS>;
  CROSS: cross specification;
  < layout properties >;

  [ cell specification; ]
}
```

In specifying the axis cross layout, the following rules must be remembered:

- The cross is specified either in the X-axis or the Y-axis, never both.
- A cell may appear either in the X-axis or the Y-axis, but never both.
- If there is a cross, both axes must be present. Cells in one axis cannot depend on cells in the other axis.

7.1.3.3 Axis Cross Layout Display

The axis cross layout defines how the information is displayed. The cells in the X-axis specification run across the rows (e.g. `X: A1, B1;` displays all the A1 then B1). The cells in the Y-axis specification run along the columns (e.g. `Y: A1, B1;` displays all the A1 then the B1). If CROSS is in the X-axis specification, the CROSS cell titles run along the rows. If it is specified in the Y-axis, then CROSS cell titles run along the columns. If CROSS is defined, but is not specified in either axis, it appears along the Y-axis by default. Cross values always appear. However, cross titles appear only if the CROSS is included in an axis specification.

In the specification `Y: A1, B1;`, even if B1 is independent of A1, all values of B1 will be displayed every time a value of A1 is displayed. The default layout behavior is to display *sparsely*. Therefore, all values of successive rows and columns are nested under the values of the preceding rows and columns. This behavior can be changed by setting the *sparse* layout property to *false*. Both behaviors are illustrated in FIGURE 60. **Note:** Cells are never nested in the cross.

FIGURE 60 Sparse Property Example

Site	Operation
S1	O11
	O12
S2	O21
	O22
	O23

Sparse set to True

Site	Operation
S1	O11
S1	O12
S2	O21
S2	O22
S2	O23

Sparse set to False

The ordering of data within the axis cross display may be controlled using the *sort* layout property. *sort* is only available in axis cross layouts. For example, the expression `sort: A1 -B1 E2;` sorts ascending first on cell A1, then descending on B1, then ascending on E2. As another example, in the *Strategy Editor*'s goals tab, the goals are sorted first by focus value, then by the name of the goal. Reducing focus on a goal moves that goal up to a higher row in the table (try this interactively!). (**Note:** This example can be found in *.../goal_list.lyt.*)

7.1.3.4 Axis Cross Layout Groups

Axis cross layouts can specify groups. If information from independent cells is to be displayed, groups allow generating a more compact display. Grouped cells are printed in a single row or column. All the values of the first cell in the group are followed by all the values of the second cell, etc. Groups may be independent or dependent.

7.1.3.4.1 Independent Groups

Independent groups have cells that are not dependent on any cell in the group. The title from each member of the group is displayed in a separate row or column prior to the group values. The syntax required for this specification is as follows:

```
(<"title cell"> group cells)
```

The title cell specifies the title to be displayed for the group values column or row. For example, refer to the file *../calendar_entry_list.lyt.*

7.1.3.4.2 Dependent Groups

In dependent groups, all cells are dependent on a different member within an independent group that precedes the dependent group in the axis specification. There is no separate column or row for the group cell titles since only group cell values are displayed. The syntax for dependent group specification is as follows:

```
[<"title cell"> group cells]
```

The title cell specifies the title to be displayed for the group values. Dependent groups are rarely used at present.

7.1.4 Tips for Working with Axis Cross

The following tips are provided for working with axis cross:

- In your worksheet, give all the cells aliases and use them. This will aid in managing the complexity of the replicating worksheet.
- Get in the habit of drawing the cell dependency graph
  - It helps you detect (and break) cycles.
  - It helps you understand compiler error messages.
  - It helps you keep things straight in your mind.

Section 8

Importing and Exporting Data

This section introduces you to exporting information from RHYTHM SCP in the form of ASCII text, importing information (without much emphasis on the GUI), and using batch-style processing. Batch-style processing is a means of using RHYTHM SCP's engine without the GUI.

Exporting is one simple way of communicating from RHYTHM SCP to other applications; the RhythmLink product line enables more sophisticated and integrated communications between RHYTHM SCP and other applications.

Importing is a form of communicating from other applications to RHYTHM SCP. Data is imported into RHYTHM SCP from data files. This imported information enables this display of information in the GUI and planning.

8.1 Export File

An export file is the layout/worksheet combination used for exporting data from the engine to ASCII files. It is similar to the import worksheets in structure and usage. The layout is called export_file layout and is based on the axis cross layout mechanism. This allows replicating worksheets to be used to store the data that is to be exported. All export files have a *.exp* extension.

8.1.1 Export File Syntax

An export file requires a specific syntax. The required fields are shown in **bold**. This syntax is shown below:

```
export_file filename
{
  < options >
  < file: pathname; >
  worksheet () {     // a replicating worksheet
    worksheet definition;
  }
  Y: cell specification;
}
```

To use an existing worksheet instead, use `worksheet: worksheet_name;`. The worksheet must be present in one of the reports directories accessible by the user.

8.1.2 Export File Options

Export file options are listed below:

- `filename: "pathname";` - the output file defaults to *filename.dat*
- `delimiter: "character";` - the field delimiter to be used in the output file
- `{before_export | after_export}: "command";` - the command to be executed before and after the export
- `fixed_width: true | false;` - If true, the output is in fixed width format
- `title_line_prefix: "character";` - the character to be used to distinguish the title line from others

8.1.2.1 Output Cells

The cell specification defines which cells from the worksheet will be exported, using the axis cross idiom. For example, in FIGURE 61 *meta_model.exp* exports to *mmnew.dat*:

FIGURE 61 Output Cells Example

```
export_file meta_model
{ delimiter: ";";
  file: mmnew;
  worksheet () {
  [A1 mt = model_types;]
  [B1 = mt.name;]
  [B2 field = mt.fields;]
  [C1 = field.type;]
  [C2 = field.name;]
  [C3 = field.description;]}
Y: B1, C1, C2, C3;}
```

For each type of model in RHYTHM SCP, for each field of that model, one record will be exported containing the name of the model. For example, Buffer, and the type, name, and description of a field (e.g. Buffer, String, description, "description of the Buffer", Buffer, Location, location, "Location of the Buffer", etc.).

8.1.3 .opt Files - Revisited

.opt files are used to specify startup option settings to scp_ui and scp_engine. The location of *.opt* files can be explicitly specified by *option_file* flag (e.g. `scp_engine option_file foo/scp_engine.opt`). scp_engine options can be used to perform startup planning activities as well as to set server parameters. scp_ui options, however, can only set various client parameters.

An example of the contents of *scp_engine.opt* and *startup.in* are shown below:

```
include: ...
...
startup: do_file("startup.in")
```

scp_engine.opt

```
import("demand");
define(PLAN, supply_chains.first.plans.first);
PLAN.site_plans.for_each(#.requests).for_each
(#.plan_to_satisfy);
do_file("other_startup_initializations");
```

startup.in

8.1.4 Export Directories

In any export directory, you can place files named *before_export.in* and *after_export.in* that allow RHYTHM SCP to do OIL pre- and post-processing upon export. For example, in the above startup.in file, upon invoking `export("demand")`, the export function first looks for a file named *before_export.in* in the *demand* directory, and processes it as `do_file("demand/before_export.in")`. It then processes all the *.exp* files. Finally, the export function looks for a file named *after_export.in* and processes it as `do_file("demand/after_export.in")`.

8.2 Import File

Import files are layout/worksheet combinations that describe how the data is to be loaded into the model. Import files have a *.imp* extension. Data files (*.dat*) contain the information that is loaded by the *.imp* files.

8.2.1 Import File Syntax

Import files require a specific syntax. All expressions must end with a semicolon. Required fields are **bolded**. The syntax is shown below:

```
import_text_file_buffers
{
   file: "buffers.dat";
   delimiter: "," ;
worksheet ( )
{
   model: "Buffer";
   sf_const Supply_Chain SUPP = find(supply_chains,
      Trios");
   sf_const Site SITE = find(SUPP.sites, "Main_Link");
   this = SITE.buffers;

   [A1: #;]
   [B1: #;]
   [C1: name = A1;]
   [D1: location = B1;]
   [E1: description = #;]
   [F1: flow_policy = #;]
   [G1: item = #;]
   [H1: producing_operation = #;]
   [H2: consuming_operation = #;]
   [I1: min_on_hand = #;]
   [J1: min_time = #;]
}
import_record: A1 B1 E1 F1 G1 H1 H2 I1 J1;
}
```

The order of evaluation for the cells is not determined by cell address. The *import_record* option specifies the order of evaluation.

8.2.2 Import File Options

Import file options are as follows:

- `file: <filename>` - the name of the *.dat* file that contains the information to be loaded by the import file (required field)
- `delimiter: <character>` - the field delimiter to use (required field)
- `model: <model name>` - the import file's target depth within the modeling hierarchy
- `sf_const` - the local constants for the import file
- `this` - the submodel list which is added to by the import file
- `import_record` - the order in which the fields are read from the *.dat* file (this field is not required; no data is read if the field is omitted)

8.2.3 Import Directories

In any import directory, you can place files named *before_import.in* and *after_import.in* that allow RHYTHM SCP to do OIL pre- and post-processing upon import. For example, in the above startup.in file, upon invoking `import("demand")`, the import function first looks for a file named *before_import.in* in the *demand* directory, and processes it as `do_file("demand/before_import.in")`. It then processes all the *.imp* files. Finally, the import function looks for a file named *after_import.in* and processes it as `do_file("demand/after_import.in")`.

8.2.4 Ports

On startup, *scp_engine* specifies a port number for network communications (TCP/IP). Prospective clients wishing to communicate with that particular *scp_engine* must specify that port number. If they are on a different machine, they must also name the host on which the scp_engine is running. If a port number is not explicitly specified, one is chosen by default (27111). If port 0 is specified, then the *scp_engine* will start, perform any planning functions specified in the startup options, and then automatically shut down. Example uses are pre-processing to create a saved memory image, or functional testing.

Appendix A Customizing the VB UI

This appendix describes a scenario whereby you add a new workbench to the UI. This scenario is provided to give you an idea of the steps you must perform in customizing the UI. You can take the information and knowledge you gain here and apply it to other customization efforts.

A.1 Overview

In this scenario, you are adding a new workbench to the UI named "Training Workbench". This workbench has one folder and the folder has two actions defined. The actions defined for this folder are:

- Bring up a new screen using OIL reports
  - Action: Launch a form to show a list of items
  - Report gives a list of items and item descriptions
  - Filters in the workbench decide which are to display
  - Right-clicks on columns of the new report allow access to standard reports
- Launch an external application
  - Action: Launch an external application (notepad.exe)

The following are the steps (which are described in detail later) that you perform during this customization scenario:

1. Write the OIL files to display Item and Item Descriptions and place these files in your *custom_reports* directory.
2. Add an action to launch "Training Workbench" to the action table.
3. Add the actions for launching the OIL reports and the external application to the action table.
4. Add the workbench details (folders and action names) to the workbench table.
5. Add "Training Workbench" to the *View* menu.
6. Add the attributes of the new report that is being launched to the attributes table.

A.2 Step 1: Add the OIL Files

The first step you must complete is writing the OIL files required to display the Item and Item descriptions. These files must be placed in your *custom_reports* directory. Examples of the *.rpt*, *.lyt*, and *.wrk* files required are shown below:

```
cust_item_list.rpt
     cust_item_list (String supply_chain_name, String site_name,
             String workbench_filter, String dummy)
     {
     variable itms = make_type(nonexistent, List[Item]);
     compute dum_itms = define(itms, if(logical(workbench_filter),
             user.selected_items,
             do(define(sc, supply_chains.find(supply_chain_name)),
             define(site, sc.sites.find(site_name)),
             site.items)));
     layout: cust_item_list(itms);
     }


cust_item_list.lyt
     axis_cross cust_item_list
     {
     worksheet: cust_item_list;
     Y: A2 A1 B2 B3 B6;
     }


cust_item_list.wrk
     replicating cust_item_list (List[Item] itms)
     {
     [A1 item = itms;
     [A2 site = item.owner;]
     [B2 = item.description;]
     [B3 = item_category;]
     [B6 = item_artificial;]
     }
```


A.2.0.1 Functional Worksheet Example

The VB UI provides an action to call functional worksheets. The functional worksheet should be placed in your report path (preferably in your *custom_reports* directory) as specified in *user.dat*. The following example explains how to create a functional worksheet and the action required to execute it.

The purpose of the following functional worksheet is to generate a plan to satisfy a request give the supply chain, plan, site and request. If * is sent through for requests and sites, then all requires in all the sites are planned. This functional worksheet is placed in your *custom_reports* directory as defined in *user.dat*.

```
function satisfy_requests(String sc_str, String pln_str,
   String site_str, String req_srt)
{
[A1 SUPP = supply_chains.find(sc_str);];
[B1 PLAN = SUPP.plans.find(pln_str);];
[C1 SITEPLANS = if(site_str == "*", PLAN.site_plans,
   PLAN.site_plans.find(SUPP.sites.find(site_str))
     .recurse(#.members));];
[D1 dummy = if(req_str == "*",
   do(SITEPLANS.for_each(#.plan_to_satisfy_all_requests),
     nonexistent), do(SITEPLANS.for_each(#.requests)
     .find(req_str).plan_to_satisfy, nonexistent));];
[F1 return_dp = SITEPLANS.first.requests.first
   .delivery_requests.first.item_requests.first
     .delivery_plan.string.echo;];
[G1 return_date = SITEPLANS.first.requests.first
   .delivery_requests.first.item_requests.first
     .delivery_plan.dates.string.echo;];
}
```

The following action, i2ExecuteSatisfyRequests, is required for executing the functional worksheet above. This action needs the following parameters set in xaction_editor:

Field	Value
Action Name	fws
Action Object	satsify_requests
Action Command Type	i2System.clsStdFws
Action Params	String context.Supply_Chain, String context.Plan, String context.Site, "*"

Now, this action can be invoked as a right-click menu item or as a workbench node.

A.3 Step 2: Add Action to Launch the Workbench

The second step you must take is adding the action to launch the "Training Workbench" to the action table. You need to assign a unique name to the action (i2DisplayWorkbench_TR in this case). To do this, use the xaction_editor. The following steps show how to display the xaction_editor and add the action to the action table:

1. At a prompt, type the following:

```
.../scp_ui port #### user "<NT userid>" initialize
'display_report(xaction_editor", "*")' &
```

   The window displays.
2. Right-click within the editor to display a popup menu.
3. Select the *New Row* option.
4. Complete the details of the action for launching the "Training Workbench"by adding the following values to the action table:

Field	Value
Action Name	i2DisplayWorkbench_TR
Action Type	form
Action Object	frmWorkbench
Action Command Type	i2System.clsStdForm
Action Params	"WorkbenchFilters_MP"
Action Title Const	Training Workbench

Note: Several of the fields that are relevant to an action for launching a report are left blank.

A.4 Step 3: Adding the Other Actions

The third step you must perform is adding the other actions necessary for launching the OIL report and the external application to the action table. To do this, perform the following steps:

1. Using the xaction_editor, complete the details for the following actions:
   - Launching the item listing/description report
   - Running the external application (Notepad.exe)
2. Add the following values for the action to launch the report:

Field	Value
Action Name	i2DisplayItemListingTWB
Action Type	form
Action Object	frmGenericReport
Action Command Type	i2System.clsStdForm
Action Params	String context.Supply_Chain, String context.site, "True", String system.time
Action OIL Rpt	cust_item_list
Action OIL Lyt	cust_item_list
Action Title Const	Item Listing
Action Tbl Name	IL
Action Context	Supply_Chain, Plan, Site

3. Add the following values for the action to launch the external application:

Field	Value
Action Name	i2DisplayNotepadTWB
Action Type	exe
Action Object	c:\WINNT35\notepad.exe
Action Command Type	i2System.clsStdExe

A.5 Step 4: Add the Workbench Details

The fourth step in this process is to add the workbench details (folders and action names) to the workbench table. To do this, follow these steps:

1. Display the xwb_editor by typing the following at a prompt:
   ```
   .../scp_ui port #### user "<NT userid>" initialize
   'display_report(xwb_editor", "*")' &
   ```
   The window displays.
2. Right-click within the Xwb Editor to display a popup menu.
3. Select the *New Row* option.
4. Add the folder "Examples" by creating a row called i2DisplayWorkbench_TR-Category-Examples. The following values must be defined for this folder:

Field	Value
WB Key	i2DisplayWorkbench_TR
WB Type	Category
WB Name	Examples

5. Add the action required to launch the report by creating a row called i2DisplayWorkbench_TR-Node-ItemListing. The following values must be defined for this action:

Field	Value
WB Key	i2DisplayWorkbench_TR
WB Type	Node
WB Name	Item Listing
WB Parent	Examples
WB Action	i2DisplayItemListingTWB
WB Action Type	form
WB Sequence	1
Action Context	Supply_Chain, Plan, Site

6. Add the node to launch the executable by creating a row called i2DisplayWorkbench_TR-Node-Notepad. The following values must be defined for this node:

Field	Value
WB Key	i2DisplayWorkbench_TR
WB Type	Node
WB Name	Notepad
WB Parent	Examples
WB Action	i2DisplayNotepadTWB
WB Action Type	exe
WB Sequence	2

A.6 Step 5: Add the Workbench to the Menu

The fifth step you must perform is adding “Training Workbench” to the *View* menu. To do this, follow the steps below:

1. Display the xmenu_actions_editor by typing the following at a prompt:

   ```
   .../scp_ui port #### user "<NT userid>" initialize'
   'display_report(xmenu_actions_editor", "*")' &
   ```

   The window displays.
2. Right-click within the editor to display a popup menu.
3. Select the *New Row* option.
4. Define “Training Workbench” as a new menu item of the *View* menu and associate an action with it by performing the following:
   1. Add a new row called “Workbench-Training Workbench”. The following values must be defined for the action of launching this workbench from the *View* menu:

Field	Value
Menu Name	Workbench
Menu Item	Training Workbench
Menu Caption	&Training Workbench
Menu Action	i2DisplayWorkbench_TR
WB Action	i2DisplayNotepadTWB

A.7 Step 6: Add the Attributes

The sixth and final step in this process is to add the attributes of the new report that is being launched in the *attrib_definitions.dat* file.

Appendix B Hints for Writing Effective OIL

The ability to write OIL effectively is important for using RHYTHM SCP software. This appendix lists some helpful hints for writing OIL effectively and efficiently.

B.1 Basic Improvements

This first section provides some basic ways in which you can improve your efficiency.

B.1.1 Minimizing List Creation

In order for you to use memory efficiently and get the best performance, put subsequent list manipulation functions inside the original function's (). For example, consider the following OIL expressions which give the same answer and where `in` is a list of plans:

```
in.for_each(#.sites).for_each(#.operations).filter(...)...

in.for_each(#.sites.for_each(#.operations.filter(...)))...
```

Which expression is better?

FIGURE 62 shows what is really happening in the first OIL expression.

FIGURE 62 List Creation Example 1

```
in.for_each(#.sites).for_each(#.operations).filter(...)...
```

For every plan in list `in`, creates a copy of the *sites* list...

...and for every site in the new *sites* list, creates a copy of the *operations* list...

...and from the new operations list, creates another list of operations!

When the datasets are large, this is clearly a time and memory hog!

FIGURE 63 shows what is really happening in the second OIL expression.

FIGURE 63 List Creation Example 2

```
in.for_each(#.sites.for_each(#.operations.filter(...)))...
```

For every plan in list in...

...retrieves the existing *sites* list for the plan, and for every site...

...retrieves the existing *operations* list for the site, and from that list, creates a new list of operations.

Obviously uses less memory than the first expression!

Also important in the minimization of list creation is refraining from filtering through lists when you know the end product. For instance, do not search through a list of operation plans if you already know the operation you need. Instead, get the list of operation plans from the operation. Therefore, rather than using the following expression:

```
site_plan.operation_plan.filter(#.operation ==
  myoperation)for_each(...)...
```

use this expression instead:

```
myoperation.operation_plans(site_plan.plan)
  for_each(...)...
```

Another element in minimizing list creation is using a previous variable rather than re-creating a similar list when you are defining more than one variable in a worksheet. For example, instead of using the following expressions:

```
foo1 = foo_plan.seller_plans.for_each
       (#.forecasts.for_each(#.entries.for_each
       (#.planned))).sum

foo2 = foo_plan.seller_plans.for_each
       (#forecasts.for_each(#.entries.for_each
       (#.committed))).sum
```


use these expressions for enhanced efficiency:

```
foo = foo_plan.seller_plans.for_each
       (#.forecasts.for_each(#.entries))

foo1 = foo.for_each(#.planned).sum

foo2 = foo.for_each(#.committed).sum
```

B.1.2 Preserving Hierarchy Information

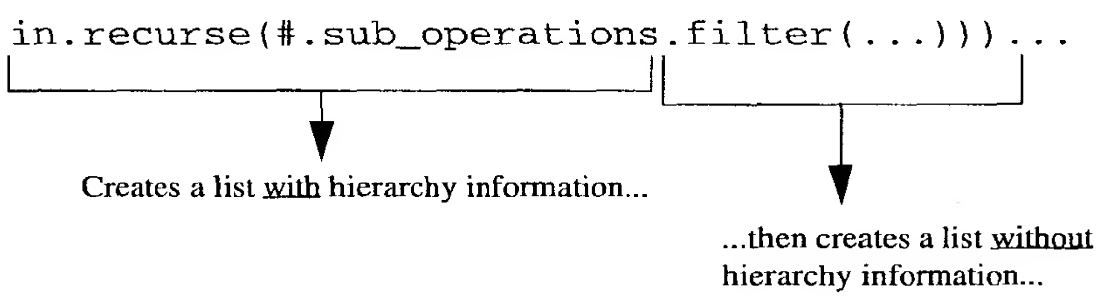
When you need to filter a list of information and you intend to show the results with a hierarchy control (e.g. *outline_general*, *indented_general*), put the `filter` expression inside the `recurse`. For example, consider the following OIL expressions, each of which generate lists with the same elements, but only one of them has *hidden* hierarchy information that certain controls can use:

```
in.recurse(#.sub_operations.filter(...))...

in.recurse(#.sub_operations).filter(...)...
```

FIGURE 64 illustrates whether or not the first OIL expression results in hierarchy information.

FIGURE 64 Preserving Hierarchy Information Example 1



This expression's result list has no hierarchy information.

FIGURE 65 shows that the second OIL expression does have hierarchy information.

FIGURE 65 Preserving Hierarchy Information Example 2

```
in.recurse(#.sub_operations.filter(...)))...
```

Creates a list without hierarchy information...

...then recursively creates a list with hierarchy information...

This expression's result list has hierarchy information.

B.1.3 Caching Values for Reuse

Cells sometimes have common sub-expressions. You can improve memory use and performance by caching the result of the common sub-expression. For example:

```
[ A1 allocation_fill_rate = plan.seller_plans.
       for_each(#.product_forecasts).for_each
       (#.entries).filter(and(#.committed > 0,
       #.committed != #.allocated)).filter
       (percentage(#.allocated/#.committed) <
       percentage(80)); ]
[ A2 allocation_consumption_rate =
       plan.seller_plans.for_each
       (#.product_forecasts).for_each
       (#.entries).filter(and(#.committed > 0,
       #.committed != #.allocated)).
       filter(and(#.allocated > 0, percentage
       (#.consumed/#.allocated) <
       percentage(50))); ]
```

The following is better because it only creates the list for the common sub-expression once:

```
COMPUTE interesting_entries =
       plan.seller_plans.for_each
       (#.product_forecasts).for_each
       (#.entries).filter(and(#.committed > 0,
       #.committed != #.allocated));
[ A1 allocation_fill_rate =
       interesting_entries.filter(percentage
       (#.allocated/#.committed) <
       percentage(80)); ]
[ A2 allocation_consumption_rate =
       interesting_entries.filter(and
       (#.allocated > 0, percentage(#.consumed/
       #.allocated) < percentage(50))); ]
```

Many times the worksheets in a report need to share information. Always try to put the shared information in the report and pass it to all the layouts that need it, instead of recomputing the same information on each tab. You may find that you can actually share the worksheets with a few small additions. This will not necessarily save you time, but will help when you are optimizing.

B.1.4 Do Not Concatenate String Values

It is important for performance to generate string values once per replication rather than concatenating the same strings over and over in loop iterations. For example:

```
[B2:PLAN.site_plans.find(SITE).
        operation_plans.filter(or(or
        (#.release_name == rel & "Prod",
        #.release_name == rel & "Cons"),
        #.release_name == rel)).for_each
        (do(#.set_ord_id(ord)id),
        #.set_qty(qty), #.set_log(TRUE),
        nonexistent));]
```

The problem with the expression above is that the same string values are created for every element of the list. To improve performance, you should use the following expression:

```
[B2: do(define(relprod, rel & "Prod"),
        define(relcons, rel & "Cons"),
        PLAN.site_plans.find(SITE).
        operation_plans.filter(or(or
        (#.release_name == relprod,
        #.release_name == relcons),
        #.release_name == rel)).for_each(do
        (#.set_ord_id(ord_id), #.set_qty(qty),
        #.set_log(TRUE), nonexistent)));]
```

Or, if *rel* does not change values during the life of the worksheet (e.g. *rel* is passed as a parameter), the following expression would provide greater performance since the strings would only be generated once:

```
variable relprod = rel & "Prod";
variable relcons = rel & "Cons";
[B2: PLAN.site_plans.find(SITE).
     operation_plans.filter(or(or
     (#.release_name == relprod,
     #.release_name == relcons),
     #.release_name == rel)).for_each(do
     (#.set_ord_id(ord_id),#.set_qty(qty),
     #.set_log(TRUE), nonexistent)));]
```

B.1.5 Getting the Index of the Current Iteration Element

When you iterate over a list, you refer to the current element as #; however, sometimes you need to know the index of the current element as well. That is, you need an OIL equivalent of a *for loop*. A pseudo-code example would be:

```
for (i = 1 to num) {
print("Load "& i &" is");
print(OPERATION.loads(i).name, ...
}
```

The OIL solution for this is to use the `integers` function and iterate over the list of integers. For example, given a list of loads for an operation:

```
[ A1 = integers(1, num).for_each("Load
     "& #.string &" is" & OPERATION.loads.
     element(#)).string("newline"); ]
```

Cell A1 will show the following string:

```
Load1 is Heavy_Load
Load2 is Light_Load
Load3 is Medium_Load
```


B.1.6 Iterating through Multiple Related Lists

When you have lists where the *n*th element in each list goes together, you can synchronize a loop over the elements by using the `integers` function. For example, given the following lists:

```
words1 ==> list("fruit", "tree", "heavy")
words2 ==> list("Apple, "Limb", "Cargo")
words3 ==> list("is a", "is part of a", "is")
```

create strings such as “Apple is a fruit”.

The OIL solution to iterating through multiple related lists is the judicious use of the `integers` and `element` functions. For example,

```
integers(1, 3).for_each(words2.
     element(#) & " " & verbs.element(#) & " " &
     words1.element(#)) ==> list("Apple is a fruit",
     "Limb is part of a tree", "Cargo is heavy")
```

Note that the code above simply assumes that *words1*, *words2*, and *verbs* all have 3 elements. To handle same-size lists of arbitrary length, you should use the `count` function.

```
integers(1, verbs.count).for_each(...)
```

B.1.7 Sorting on Multiple Keys

The `sort` function is simple to use. But what if you want a list sorted by more than one key? The OIL solution is as follows:

```
sort(if(a.primary_key != b.primary_key,
     a.primary_key < b.primary_key,
     a.secondary_key < b.secondary_key));
```

For example, sort a list of operation plans by start dates (primary key). If any of them start on the same day, sort those operations alphabetically (secondary key). The OIL expression is as follows:

```
SITEPLAN.operation_plans.sort(if
     (a.operation.date != b.operation.date,
     a.operation.date < b.operation.date,
     a.dates.name < b.dates.name));
```

B.1.8 Empty Initial Values

Variables are normally initialized with meaningful data. However, sometimes you need to initialize a variable to an empty list or *nonexistent*. This causes problems when you need to write a cell expression that relies on the type of the value. You should use `make_type` to circumvent this problem.

FIGURE 66 shows an example of an expression that the compiler will not like and one that the compiler will like:

FIGURE 66 Empty Initial Values Example

```
VARIABLE chosen_operation = nonexistent;
[ A1 = chosen_operation.sub_operations; ];
```

Since the type of *chosen_operation* is unknown, the compiler cannot detect which *sub_operations* function to use.

```
VARIABLE chosen_operation = make_type(nonexistent, Operation);
[ A1 = chosen_operation.sub_operations; ];
```

The type of *chosen_operation* is specified here, and the compiler can detect which *sub_operations* function to use.

B.1.9 Symbols vs Strings

A string is a simple vector of characters. Strings are not shared. If the same string is used 50 times within RHYTHM SCP, it takes 50 times as much memory as one string. Most description fields in the *RHYTHM SCP Model Reference Manual* are strings because they are rarely shared.

A symbol is a shared string. Symbols are interned in a hash-table, so if the same symbol is used 50 times with RHYTHM SCP, it only takes up storage equal to one string (plus 3 words of overhead). Another advantage of symbols is that equal symbols have the same address in memory, which makes the "==" and "!=" operators for symbols much faster. Most model names in the *RHYTHM SCP Model Reference Manual* are symbols, to avoid duplication and make comparison faster.

The distinction between strings and symbols is important when deciding on the data-type for a user-defined field.

- Use symbol where you can take advantage of sharing
- Use string when the contents are probably unique

Avoid converting symbols to strings in the UI when you do not need to. For example,

```
list_of_buffers.for_each(#.string);
                    //makes a list of name STRINGs
list_of_buffers.for_each(#.name);
                    //makes a list of name SYMBOLs
```

B.1.10 for_each

Do not make *for_each* work harder than necessary. For example,

```
lists.for_each(#.listb).for_each(#.listc)
```

takes longer than

```
lists.for_each(#.listb.for_each(#.listc))
```

The time is computed for every layer of nesting.

B.1.11 if

You should reduce the logical computation of `if` statements where possible. For example,

```
[B4 = ATP_SP.operation_plans.filter
     (#.released == True).filter(#.motive ==
     "DELIVER").filter(#.item.name ==
     A2.name).for_each(#.units).
     sum ? number("0");]
```

is better than:

```
[B4 = if(foo.foo_plans.filter(#.released ==
     True).filter(#.motive == "DELIVER").
     filter(#.item.name == A2.name).
     for_each(#.units).sum.exists.not,
     number("0"), foo.foo_plans.filter
     (#.released == True).filter(#.motive ==
     "DELIVER").filter(#.item.name == A2.name).
     for_each(#.units).sum);]
```

B.1.12 filter and sort

You should always perform `filter` before `sort`. As an example, you should use the following expression:

```
foo = foo1.filter(foo2).sort(a.sort_qty <
     b.sort_qty);
```

rather than this expression:

```
foo = foo1.sort(a.sort_qty < b.sort_qty).
     filter(foo2));
```


B.1.13 Do Not Compute Unnecessarily

Reports have to react to changes, but you can minimize the recomputing that takes place. For example, you could use:

```
compute a = plan.site_plans.
     for_each(#.requests);
```

However, the following OIL code is much more efficient:

```
variable current_plan = plan;
variable a = plan.site_plans.
     for_each(#.requests);
compute check_plan = if(current_plan == plan,
     current_plan, do(set_variable(a,
     .site_plans.for_each(#.requests)),
     set_variable(current_plan, plan)));
```

Then, the list will only be recomputed if the plan changes.

Do not compute something, then discard it with an `if`. Instead, check the condition up front to avoid the computation. For example, some reports have options to select what RHYTHM SCP displays.

Do not compute the intermediate results if the final result is known to be *nonexistent*. Also, do not loop through a large list filtering out something, then loop around this inner loop using a different filter criterion. Instead, loop through the list once gathering together the different data in a single pass. This is what `bucketize` does, for instance.

The following examples show a poor way and a better way to compute information:

```
foo = foo1.sort(a.sort_qty < b.sort_qty).
     filter(foo2))

foo = foo1.filter(foo2).sort(a.sort_qty <
     b.sort_qty)
```

In the set of examples above, the second expression is much more efficient to the first expression.

```
[ B4 = if(foo.foo_plans.filter(#.released ==
     True).filter(#.motive == "DELIVER").
     filter(#.items.name == A2.name).for_each
     (#.units).sum.exists.not, number("0"),
     foo.foo_plans.filter(#.released ==
     True).filter(#.motive == "DELIVER")
     .filter(#.items.name == A2.name).
     for_each(#.units).sum); ]

[ B4 = ATP_SP.operation_plans.filter
     (#.released == True).filter(#.motive ==
     "DELIVER").filter(#.items.name ==
     A2.name).for_each(#.units).sum ? number
     ("0"); ]
```

In the set of examples above, the second expression if more efficient than the first.

B.1.14 And, Or are Short-Circuits

The logical operators `and` and `or` perform short-circuit evaluations. Once the `and` operator finds a value that returns false, it returns false and stops evaluating. Once the `or` operator finds a value that returns true, it returns true and stops evaluating. So, you can gain some speed wins by using `and` and `or` when you filter.

For example, using the expression from previous section you can improve its efficiency by adding the `and` operator:

```
[ B4 = ATP_SP.operation_plans.filter(and
     (#.released == True).filter(#.motive ==
     "DELIVER").filter(#.items.name ==
     A2.name).for_each(#.units).sum ? number
     ("0"); ]
```

In general, you should:

- Put the clause that will return *false* most often first in the `and` list
- Put the clause that will return *true* most often first in the `or` list

B.1.15 Using bucketize

You should use the *bucketize* function to gather different data from a large list rather than looping through a huge list filtering for some data, then looping around this inner loop using different filter criteria. As an example, consider the following expression:

```
compute buckets = msp_seller.
     seller_plan(plan).product_forecasts.
     for_each(#.entries).bucketize(bucket_list,
     #.delivery_dates.start, #.owner.
     product.name);
```

B.1.16 Finding Problem Areas

If you have a worksheet that seems to be taking a long time to come up, try running the scp_engine with the *timings* option. Given a number as a parameter (e.g. `timings 50`) the *timings* option will print out the calculations in the worksheet that take longer than 50 milliseconds. This will aid you in finding the cells and expressions that are causing problems.

B.2 Performance Tuning

Several performance problems can occur when using OIL, such as:

- reports coming up too slowly
- reports updating too slowly
- import files processing too slowly
- batch files running too slowly
- UI response time too slow

These problems are caused by bottlenecks such as:

- excessive use of computes
- not separating common subtasks
- expressions not calculated by underlying code

B.2.1 Using Option Files

One way to improve the engine's performance is to use option files. Option (*.opt*) files allow you to track problems and to optimize the engine. The following is an example *scp_engine.opt* file:

```
scp_engine.opt
   seed:                              1
   command_execute:                   true
   action:                            true
   diagnostic:                        true
   absolute_pathnames:                true
   stdout:                            true
   dont_dispose_com_var_val:          true
   debug:                             true
   timing:                            200
   memory:                            1000000
   command_usage:                     last queue elapsed, last
                          run elapsed, last reply elapsed
   command_usage_file:                -
```

The *command_execute* option returns every command (time, user, etc.). You can track problems easier using the last four options. The *dont_dispose_com_var_val* option allows you to keep computed values. The *debug* option is the OIL debugger. The *timing* option is set to milliseconds and the *memory* option is set to bytes. The *command_usage* option keeps a log of the specified items in milliseconds. The *command_usage_file* echoes to stdout.

Note: Refer to the section Engine and UI Options on page 26 for more information on *.opt* files and a list of all engine and UI options.

B.2.2 Timing and Memory

The *timing* and *memory* engine options are used to track performance. They report engine time only (only the time it takes on the CPU). The options must be used together. If you use only one, the engine returns every memory allocation. **Note:** When using these options, the dependent times are cumulative so you have to subtract to get the exact value for a particular expression.

B.2.3 Mem_metrics

Mem_metrics tracks exactly what kinds of memory are allocated and where they are allocated. This allows you to look at a "snapshot" of memory allocations and see what is taking a large amount of memory and what is not. The options for mem_metrics are:

- mprof_metrics - displays memory usage after the process has allocated some amount of megabytes
- default_metrics_percentage - defines how much of the allocated memory to display (if no specification, returns 80%)
- mprof - turns on the recording of memory usage information
- mprof_log - provides a "dump" of memory usage information to a file in order to produce a graph

The following is an example of an option file (*scp_engine.opt*) using mem_metrics:

```
tips_test \
   option_file scp_engine.opt \
   open x31.saved \
   startup_file start21675.in \
   default_metrics_percentage 100 \
   mprof_metrics 10 \
   mprof 1 \
   mprof_log blp_plan_x44.memory.log \
   system $I2_HOME/tests/blp/system \
   absolute_pathnames \
   port 5619 \
   +stdout \
   | tee 21675.log
```

The | *tee 21675.log* command echoes the stdout, but rather than running in an emacs shell, this outputs to a file that you can bring up in emacs and compare the result.

When trying to find a memory growth problem, use the *mem_all_in_use* option. This option prints information for every piece of memory it can track.

B.2.4 Common Pitfalls

Some common pitfalls to try to avoid when using OIL are:

- redoing subtasks instead of storing the result
- overlooking the case where a value is *nonexistent*
- using *if* instead of the existence operator "?"
- using list operations out of optimal sequence (filter, for_each, sort)
- refiltering instead of using booleans (filter using and instead of using multiple filters)

Some time consuming expressions are:

- operation.super_operations - very slow because it does not cache
- buffer.all_consuming_operations and buffer.all_producing_operations
- find_or_create - prepares to create before finding
- contains - cannot make any assumptions about a list it is working on so it does an exact instance match
- date arithmetic - if you must use this expression, set the time at the top of the file rather than in the middle of a complex computation

Appendix C OIL Debugger

i2 has created a debugger for use with OIL. This section provides an overview of OIL Debugger, the functions associated with it, and some example uses of it.

C.1 OIL Debugger Overview

OIL Debugger is a simple interface and uses an OIL Listener. When stopped at a breakpoint, an OIL Listener is entered. The OIL Listener reads from scp_engine's standard-in and writes to standard-out. You will see an "ODB>" prompt when OIL Debugger is active. You can enter any OIL expression at the prompt and see the evaluation results. **Note:** Entering an empty line causes the previous command to run again. This is handy for running the *next*, *step*, or *cont* functions repeatedly.

C.2 OIL Debugger and scp_engine

You cannot run scp_engine in the background and use these debugging aids. You can run scp_engine in an emacs shell buffer instead. This way you can use meta-p to recall previous commands, you get an infinite history, and it is easy to cut and paste. **Note:** All breakpoints default to disabled if the debug option is FALSE (which is the default). Breakpoints default to enabled when the debug option if TRUE. Breakpoints can be controlled through the Breakpoint model. See the *enable* and *disable* functions.

C.3 OIL Debugger Functions

The following functions were added to OIL for debugging purposes. **Note:** Optional parameters are in square brackets [ ].

- break([expression, [, name [, condition]]])
- breakpoint([, name])
- stop(cell_name [, worksheet_name])
- enable[(breakpoint_name)]
- disable[(breakpoint_name)]
- next[(count)]
- step[(count)]
- cont[(count)]
- watch([expression [, name [, condition]]])
- unwatch(watchpoint_name)
- where
- whereis
- print_variables
- print_report_variables
- inspect(any_model)

C.3.1 break Function

The *break* function (`break (Expression result, Symbol name, Expression condition)`) executes the first parameter, returning its value. It also creates a Breakpoint model for the expression.

If no parameters are specified, the current list of breakpoints is printed. The first parameter can be of any type, and is the result from this function. The second parameter is the optional breakpoint name (the default is a small integer). **Note:** There can be multiple breakpoints with the same name. All are enabled and disabled together, have the same condition, etc. The third parameter is an optional breakpoint condition which must evaluate to TRUE before breaking. *condition* is evaluated before *expression.*

C.3.2 breakpoint Function

The *breakpoint* function either creates a breakpoint or finds a breakpoint with a particular name. For example,

```
breakpoint()               // creates a breakpoint

breakpoint(Symbol name) // finds or creates a
                        // breakpoint with the
                        // name "name"
```

C.3.3 stop Function

The *stop* function either creates a breakpoint for the named cell and worksheet or creates a breakpoint for the named cell in the current worksheet. See the *break* function for more details. For example,

```
stop(Symbol cell_name, Symbol worksheet_name)
            // creates a breakpoint for the named
            // cell and worksheet

stop (Symbol cell_name) // creates a breakpoint for
                        // the named cell in the
                        // current worksheet
```

C.3.4 enable Function

The *enable* function enables the specified breakpoint or all breakpoints. For example,

```
enable()         // enables all breakpoints

enable(name)     // enables the breakpoint
                 // named "name"
```

C.3.5 disable Function

The *disable* function disables the specified breakpoint or all breakpoints. For example,

```
disable()        // disables all breakpoints

disable(name)    // disables the breakpoint
                 // named "name"
```

C.3.6 next Function

The *next* function runs until the next cell, do_file line or variable assignment. It does not cross into other call or do_file files. (See *step* for more details.) When you use the parameter count, it specifies the number of breakpoints to skip before actually stopping.

C.3.7 step Function

The *step* function runs until the next cell, do_file line or variable assignment. It will step into other call or do_file statements. (See *next* for more details.) When you use the parameter count, it specifies the number of breakpoints to skip before actually stopping.

C.3.8 cont Function

The *cont* function continues until the next breakpoint. When you use the parameter count, it specifies the number of breakpoints minus one to skip before actually stopping.

C.3.9 watch Function

The *watch* function creates a watchpoint model object that executes the expression and prints the results at every breakpoint executed in the same context as *watch* was executed in (the same worksheet or do_file).

If no parameters are specified, the current list of watchpoints is printed. The first parameter can be of any type, and is the result from this function. The second parameter is the optional watchpoint name (the default is a small integer). **Note:** If there is already a watchpoint with the given name, its expression and condition are replaced with new definitions. The third parameter is an optional watchpoint condition which must evaluate to TRUE before printing. *condition* is evaluated before *stuff.*

If the third *condition* parameter is `nonexistent` (the default) the watchpoint prints only when the results are different than the last time. To print every time, use TRUE for the *condition*.

C.3.10 unwatch Function

The *unwatch* function removes the named watchpoint model object (created by *watchpoint*). Its first parameter is the name of the watchpoint to delete. If the name is *all*, then all watchpoints are deleted.

C.3.11 where Function

The *where* function returns a one-line description of where this function is executing.

C.3.12 whereis Function

The *whereis* function prints a detailed description of where this function is executing.

C.3.13 print_variables Function

The *print_variables* function prints all the variables and their values.

C.3.14 print_report_variables Function

The *print_report_variables* function prints all the report variables and their values.

C.3.15 inspect Function

The *inspect* function allows you to inspect any of the following:

- any model, displaying all its fields and sub-models
- any list
- all extension selectors
- all extended fields
- all user-defined fields

You can drill down into the model using the *inspectS*, *inspectH* or *inspectHS* functions.

The *inspect* function's parameters include any model, or list[model]. It allows you to drill down into a field (or list of fields) displayed by the previous *inspect** call.

The *inspect* function numbers the fields it prints. The *inspectS* function (`inspectS (int item_number)`) takes one of these numbers as its parameter. The *inspectH* function (`inspectH (int history_number)`) recalls a previously inspected value. Each time one of the *inspect** functions is called, the first line shows a history number. Using that number as its parameter, *inspectH* prints the same inspection again. The *inspectHS* function (`inspectHS (int history_number, int field_number)`) allows you to drill down into a field (or list of fields) of a previously inspected value. *inspectHS* uses the numbers from *inspectH* and *inspectS* as its parameters.

The *inspectV* functions are for use with calling other OIL functions on values that have previously been inspected. The first *inspectV* function (`inspectV (int field_number)`) returns the value of a field from the last inspected model or list. It uses a number (derived from *inspect* numbering the fields it prints) as a parameter. The second *inspectV* function (`inspectV (int history_number, int field_number)`) returns the value of a field from the last inspected model or list. This version of the *inspectV* function uses the same parameters as the *inspectHS* function.

C.3.16 Customizing Debugger Prompt

The OIL Debugger prompt can be customized using the *debug_prompt* option. For example:

```
set_option("debug_prompt", "Break {0}>")
```

This expression will use the breakpoint name in the prompt (e.g. "`Break 3>`").

C.3.17 Setting Breakpoints in startup.in

To set a breakpoint in your *startup.in* file, you must manually insert the break function in your OIL code.

C.3.18 Setting Breakpoints in do_file

To set a breakpoint in a do_file, edit the file to add a *break* function. For example:

```
if(or(exact(st_str,"*"),exact(end_str,"*")),
   define(d_r,PLAN.horizon).break,
   define(d_r,date_range(date(st_str),date(end_str)))
);
```

or:

```
if(or(exact(st_str,"*"),exact(end_str,"*")),
   define(d_r,PLAN.horizon).break("some_name"),
   define(d_r,date_range(date(st_str),date(end_str)))
);
```

or:

```
if(or(exact(st_str,"*"),exact(end_str,"*")),
   define(d_r,PLAN.horizon).break("some_name",
     site=="foo"),
   define(d_r,date_range(date(st_str),date(end_str)))
);
```

In the first example, a breakpoint is created by adding *.break*. In the second example, a named breakpoint is added. In the third example, a conditional breakpoint is added.

C.3.19 Disabling/Enabling Breakpoints

To disable a breakpoint, type the following:

```
disable("name")
```

where *name* is the name of the breakpoint. If *name* is **all**, then all breakpoints are disabled.

To enable a breakpoint, type the following:

```
enable("name")
```

where *name* is the name of the breakpoint. If *name* is **all**, then all breakpoints are enabled.

Appendix D

UI Design Principles and Philosophy

There are several things to keep in mind when designing your UI depending on what you are trying to accomplish. This appendix attempts to list some of the more important general guidelines to follow:

- Keep the system's behavior consistent. The same class of object should generate the same type of feedback and resulting behavior, not matter where it appears. For example,
  • a *checkbox* always changes the value of the toggle; it may invoke side effects associated with the value that has changed (e.g. changing a site from *managed* to *unmanaged*), but it does **NOT** pop up a list of choices like a *combo_popdown* or move the window like a scrollbar.
  • a scrollbar always scrolls the contents of a window; it does **NOT** pop up a dialog or flash the screen.
  • a user presses a *button* to make something happen; the *button* never "presses itself" in response to some internal function call, or acts like a *checkbox*.
- Pick the appropriate control for the task at hand. Deciding can be tricky when controls do similar things (e.g. *combo_popdown* and a group of *radio_buttons* both select one value from a small set of known values).
  • If a cell is editable and possible values are a small set (<10) and known ahead of time, a group of *radio_buttons* is a good choice.
  • If a cell is editable and possible values are limited (<15), then a *combo_popdown* is a good control.
  • If the number of possible values is more than a few and dynamic, use mechanisms to permit the user to choose instead of typing (such as *model_choose*).
- By design a control does only one thing. For example, a label is designed to display text using a certain style. So adding a select action to a label that causes it to bring up a report is making that label behave as if it were a *button*; this is surprising and unintuitive behavior to the user.
- Buttons *do* something; that is, they make the software actually perform some function.

- Flashy colors are great for demos but absolutely horrible for day-to-day use.
  • Highly saturated colors are easiest to see, and hardest to look at for long periods of time.
  • Use 3 to 4 colors at most (charts, graphs, and images are exceptions).
- Assume that errors will happen and provide ways to negotiate them.
- Avoid cascading menus. IF you must cascade menus, only put items that are used less often in the submenus, and **never use** more than 3 levels.
  • The ergonomics involved are too tricky, and puts the user into mental gymnastics to remember the desired UI item's location.
- Help the user know what data is being displayed in a report; include a context header.
  • Many are provided in the standard reports directory; reuse when possible, and follow their style if you have to create a new one.
- Important functionality should never have a hidden mechanism as the only means of invoking it.
  • For example, since the right-click popup menu is a hidden mechanism, anything that is available in that menu should also be available through regular menus in the menubar.
- Minimize the learning curve.
  • Use the same names/terms/phrases throughout the interface. If you call it a *demand channel* in one place, call it a *demand channel* in all places.
  • To the extent that is feasible, use industry and/or customer-specific terminology.
  • Make use of the common UI idioms you have become familiar with as a 1990's computer user (e.g. *Save* always goes under the *File* menu, and dialogs always have a *Cancel* or *Close* button).

Appendix E List of Images

The following is a list of images provided in RHYTHM SCP. Note that these are subject to change without notice.

add_column	find_down
add_row	find_down_right
alternate	find_left
around_left	find_left_up
around_over	find_right
around_right	find_right_down
around_under	find_up
bar_chart	find_up_left
buffer	flow
buffer_in	forecast
buffer_out	freeze
calendar	freeze_report
choose	gantt_chart
close	help
close_report	hide_toolbar
copy	is_medium
cut	import
date_time	indented_list
edit	invalid
error	item
export	juggler_large
filter	juggler_small

Appendix F

Axis Cross Examples

This appendix provides examples of axis cross layouts.

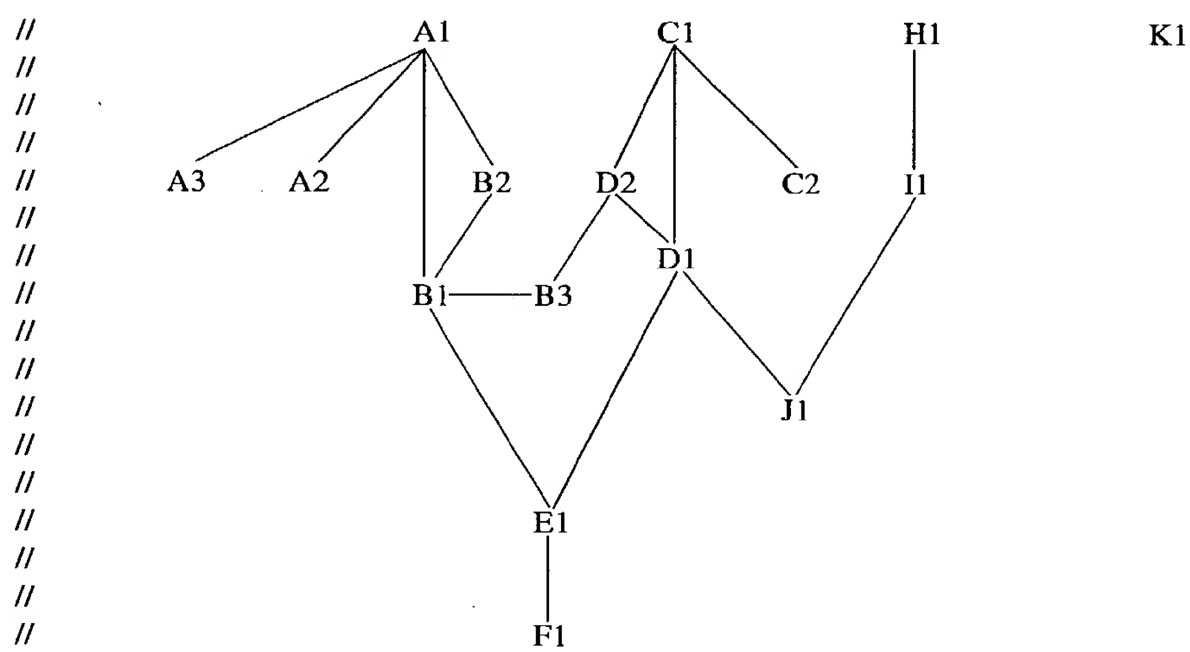
F.1 Common Worksheet Definition

The following worksheet definition is used in most of the axis cross examples that follow.

Disclaimer: All of these examples were lifted directly from internal tests; that means the data are contrived.

```
replicating cross ()
{
   variable alist = list("a", "b", "c");
   variable blist = list("1", "2", "1", "2", "3", "1", "2");
   variable clist = list("x", "y");
   variable dlist = list("7", "8", "9", "7", "8");
   variable elist = list("7", "8", "9", "nonexistent", "8");

// Cell Dependency Graph
```




```
   [ A1 resource = alist; ];// Independent
   [ A1.title = "A1"; ];
   [ A2 = A1; ];
   [ A2.title = "A2"; ];
```

```
  [ B2 alts = if(A1=="A", blist.sublist(1,2),   // F(a1)
                      if(A1=="b", blist.sublist(3,3),
                           blist.sublist(6,2))); ];
  [ B2.title = "B2"; ];
  [ B1 = A1 & B2; ];
  [ B1.title = "A&B" ];
  [ C1 = clist.sublist(1,2); ];          // Independent
  [ C1.title = "C1"; ];
  [ C2 = C1; ];
  [ C2.title = "C2"; ];
  [ D2 = if(C1=="x", dlist.sublist(1,2),        // F(c1)
                                 dlist.sublist(3,3)); ];

  [ D1 = C1 & D2; ];
  [ D1.title = "C&D"; ];
  [ E1 = B1 & D1; ];
  [ E1.title = "B&D"; ];
  [ F1 = upper(E1); ];
  [ F1.title = "upper(E)"; ];
  [ G1 = elist; ];
  [ G1.title = "Range Test"; ]; // Tests copying of ranges,
                                   and ranges with nonexistent
                                   members
  [ H1 = list("").sublist(2,2); ]; // Empty list
  [ H1.title = "H1"; ];
  [ I1 = H1; ];
  [ I1.title = "I1"; ];
  [ J1 = if(exists(I1), "exists-", "nonexists-") & D1; ];
  [ J1.title = "I & D"; ];
  [ K1 = "Doh!"; ];            [ K1.title = "Homer"; ];
  [ K2 = nonexistent; ];       [ K2.title = "Nonex2"; ];
  [ K3 = nonexistent; ];       [ K3.title = "Nonex3"; ];
}
```

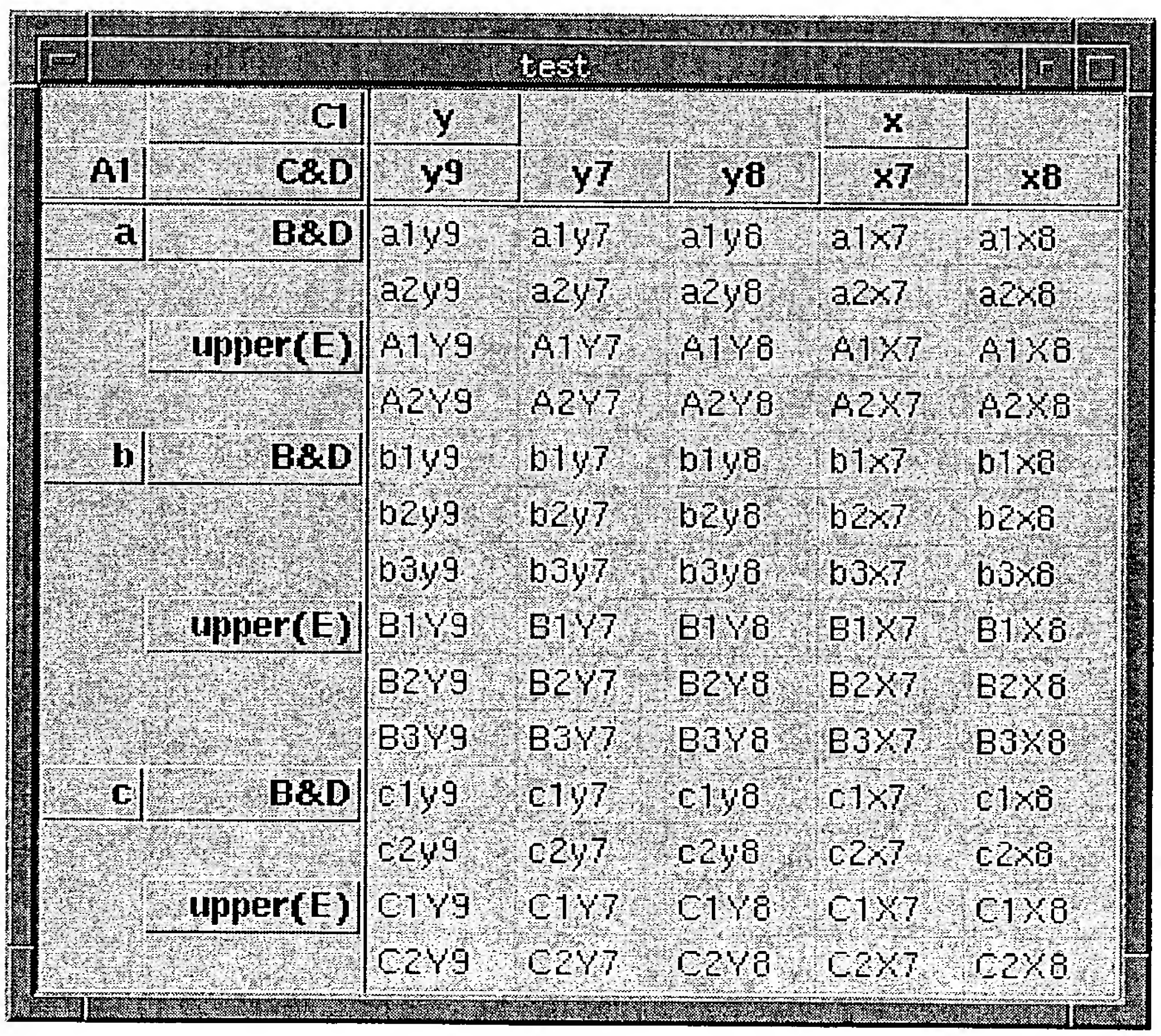

F.2 Example Set A

The following sections illustrate axis cross layouts in several formats.

F.2.1 Example Layout 1

This example shows a fairly typical axis cross with items in the X, Y and CROSS. In this case, the cross is in the Y-axis.

```
axis_cross cross_test
{
  no_scroll: true;
  worksheet: cross;
  sort: -c1;
  cross: E1, F1;
  X: c1, d1;
  Y: a1, cross;  // Leave out B1 to test the case where a
                 // replicator that doesn't compute a list
                 // is not displayed, but it's a
                 // dependency of something that
                 // IS displayed.
}
```



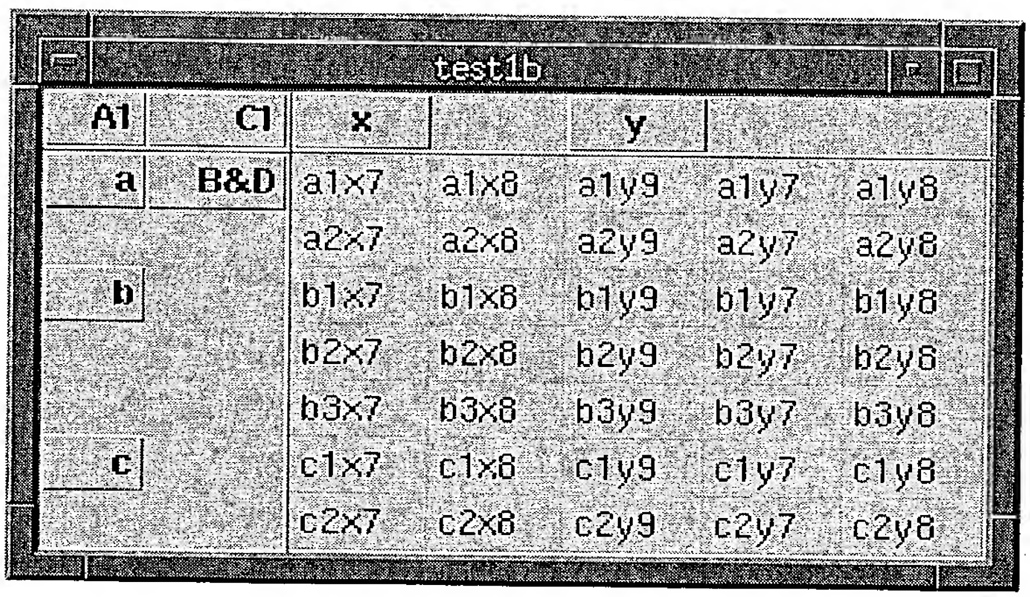
test

	C1	y			x	
A1	C&D	y9	y7	y8	x7	x8
a	B&D	a1y9	a1y7	a1y8	a1x7	a1x8
		a2y9	a2y7	a2y8	a2x7	a2x8
	upper(E)	A1Y9	A1Y7	A1Y8	A1X7	A1X8
		A2Y9	A2Y7	A2Y8	A2X7	A2X8
b	B&D	b1y9	b1y7	b1y8	b1x7	b1x8
		b2y9	b2y7	b2y8	b2x7	b2x8
		b3y9	b3y7	b3y8	b3x7	b3x8
	upper(E)	B1Y9	B1Y7	B1Y8	B1X7	B1X8
		B2Y9	B2Y7	B2Y8	B2X7	B2X8
		B3Y9	B3Y7	B3Y8	B3X7	B3X8
c	B&D	c1y9	c1y7	c1y8	c1x7	c1x8
		c2y9	c2y7	c2y8	c2x7	c2x8
	upper(E)	C1Y9	C1Y7	C1Y8	C1X7	C1X8
		C2Y9	C2Y7	C2Y8	C2X7	C2X8

F.2.2 Example Layout 2

This layout shows an axis cross without a scroll bar and the cross defaults to the Y-axis.

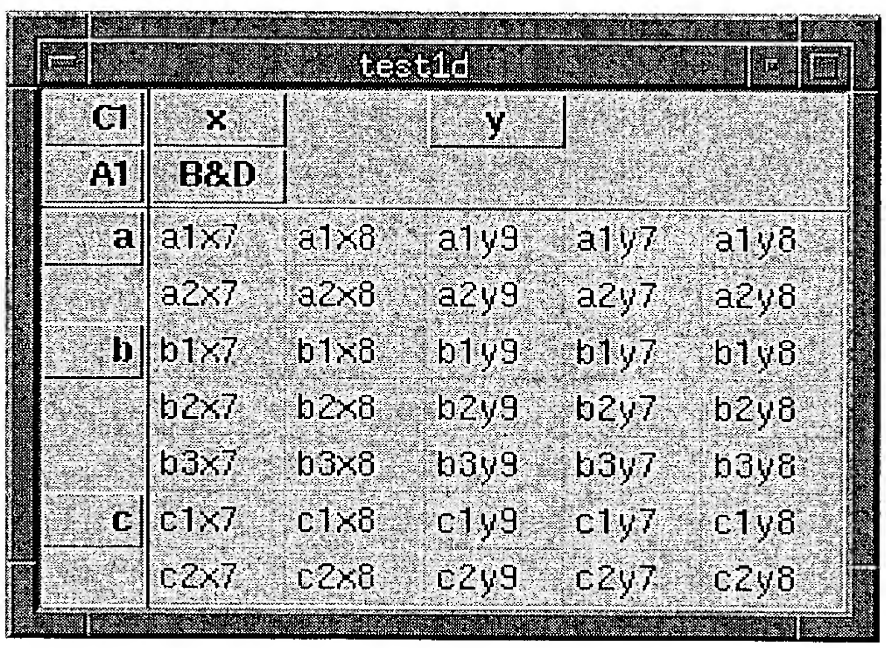
```
axis_cross cross_test1b
{
no_scroll: true;
worksheet: cross;
X: c1;
Y: a1;
cross: E1;
}
```



F.2.3 Example Layout 3

This example shows an axis cross with the cross displayed in the X-axis and no scrollbar.

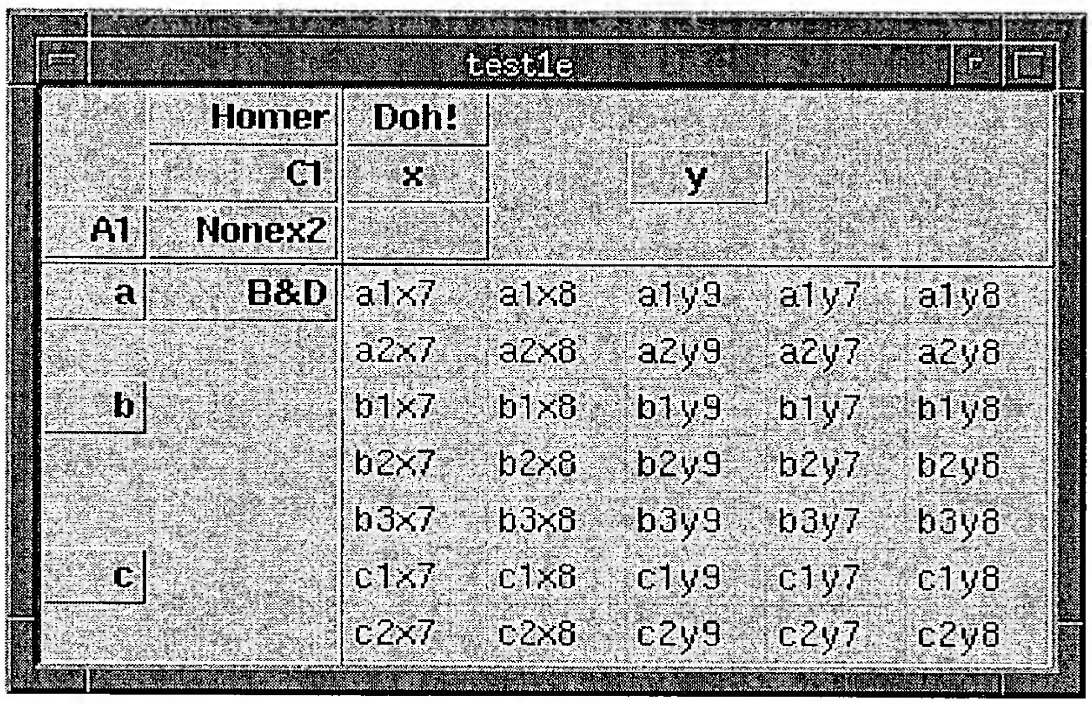
```
axis_cross cross_test1d
{
no_scroll: true;
worksheet: cross;
X: c1 cross;
Y: a1;
cross: E1;
}
```



F.2.4 Example Layout 4

This axis cross example has items in the X, Y and CROSS. The cross is displayed in the Y-axis, and there is no scrollbar.

```
axis_cross cross_test1e
{
no_scroll: true;
worksheet: cross;
X: k1 c1 k2;
Y: a1 cross;
cross: E1;
}
```



F.2.5 Example Layout 5

This axis cross contains items in the X, Y, and CROSS. The cross displays in the Y-axis, there is no scrollbar displayed, and *sparse* is set to *false* (see the section on *Axis Cross Display* in *OIL Lesson 5*).

```
axis_cross cross_test2
{
no_scroll: true;
worksheet: cross;
X: c1, d1;
Y: a1, B1 cross;
cross: E1;
sparse: false;
}
```

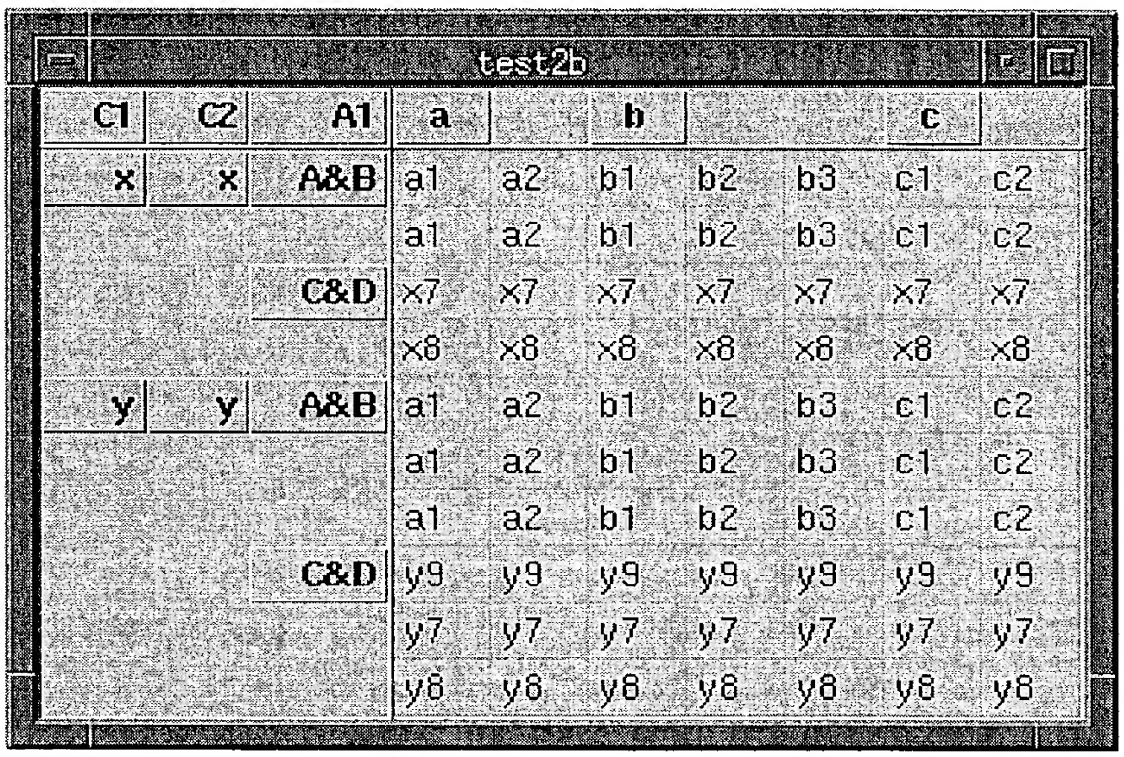
test2

		C1	x	x	y	y	y
A1	A&B	C&D	x7	x8	y9	y7	y8
a	a1	B&D	a1x7	a1x8	a1y9	a1y7	a1y8
a	a2	B&D	a2x7	a2x8	a2y9	a2y7	a2y8
b	b1	B&D	b1x7	b1x8	b1y9	b1y7	b1y8
b	b2	B&D	b2x7	b2x8	b2y9	b2y7	b2y8
b	b3	B&D	b3x7	b3x8	b3y9	b3y7	b3y8
c	c1	B&D	c1x7	c1x8	c1y9	c1y7	c1y8
c	c2	B&D	c2x7	c2x8	c2y9	c2y7	c2y8

F.2.6 Example Layout 6

This example of axis cross has items in the X, Y, and CROSS. In this case, the cross is shown in the Y-axis.

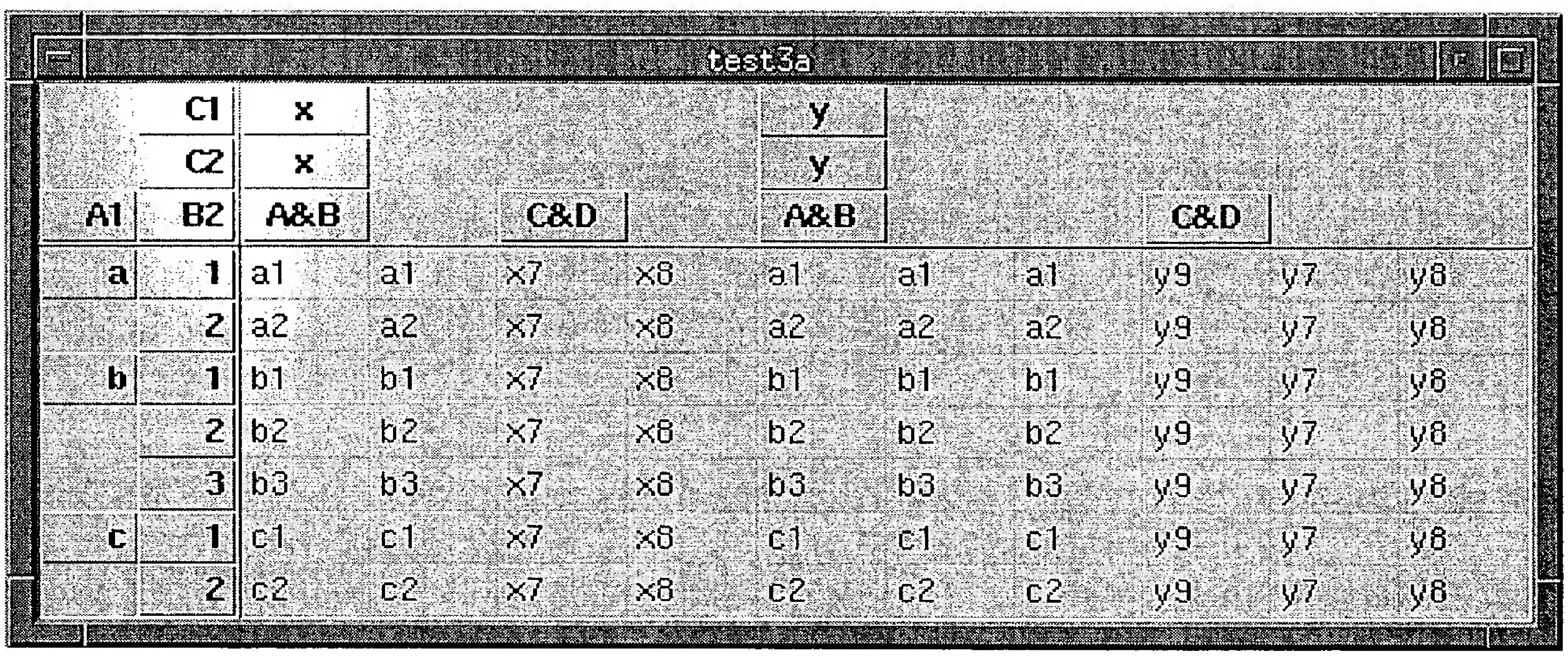
```
axis_cross cross_test2b
{
no_scroll: true;
worksheet: cross;
y: c1 c2 cross;
x: a1;
cross: b1 d1;
}
```



F.2.7 Example Layout 7

This axis cross example contains items in the X, Y, and CROSS. The cross is shown in the X-axis along with two other elements. No scrollbar is present.

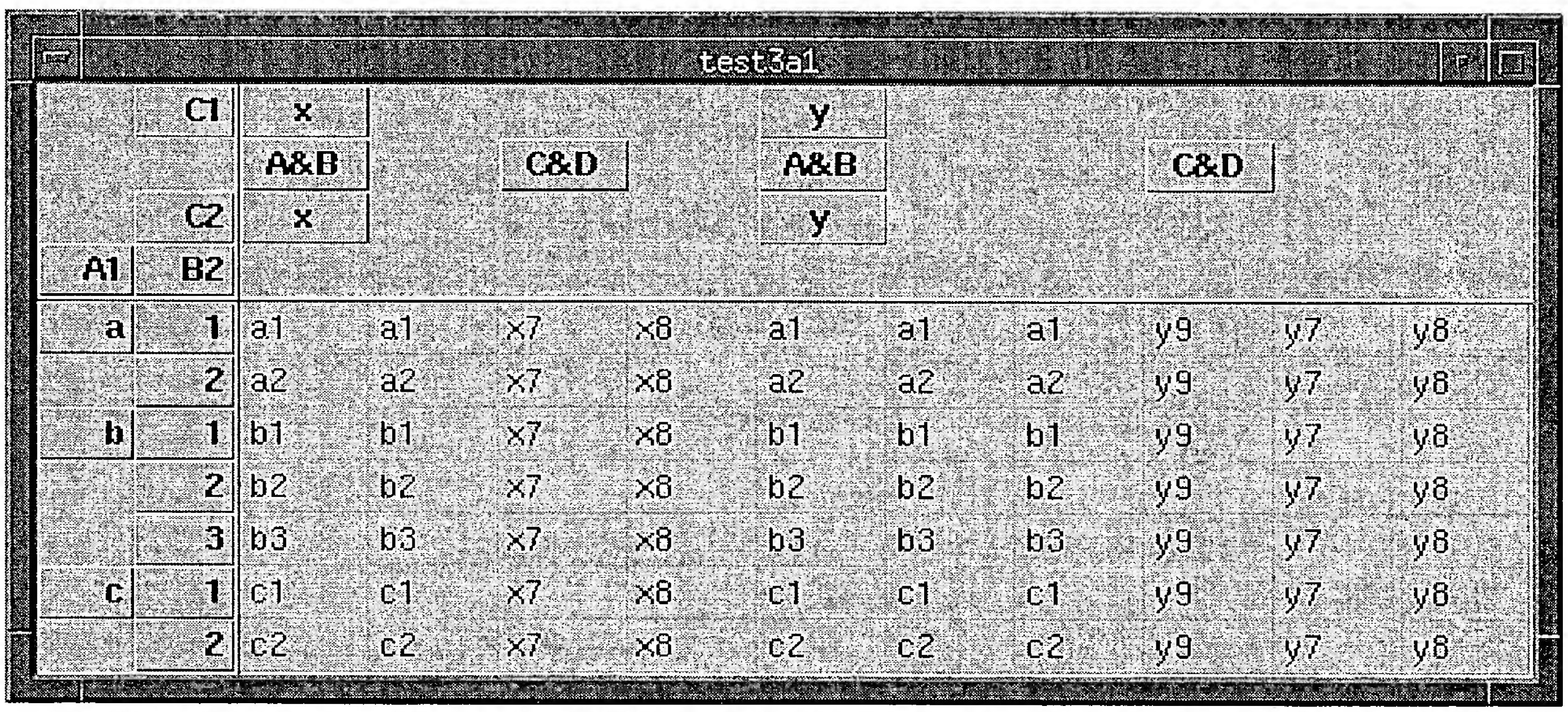
```
axis_cross cross_test3a
// Cross and 2 elements in the x axis
{
no_scroll: true;
worksheet: cross;
x: c1 c2 cross;
y: a1 b2;
cross: b1 d1;
}
```



F.2.8 Example Layout 8

This example of axis cross also contains two elements and the CROSS in the X-axis. Again, no scrollbar is present.

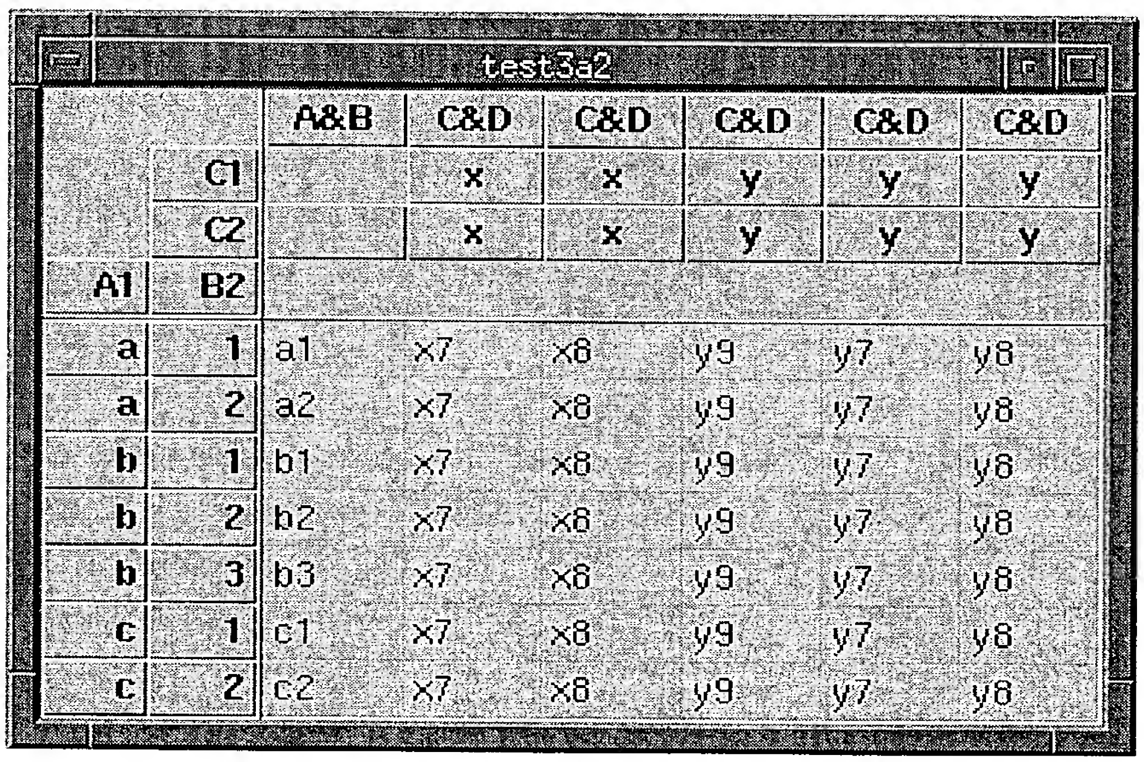
```
axis_cross cross_test3a1
// Cross and 2 elements in the x axis
{
worksheet: cross;
no_scroll: true;
x: c1 cross c2;
y: a1 b2;
cross: b1 d1;
}
```



F.2.9 Example Layout 9

The following axis cross example contains items in the X, Y, and CROSS. The X-axis holds two elements and the CROSS. No scrollbar is present and *sparse* is set to *false* (see the section on *Axis Cross Display* in *OIL Lesson 5)*.

```
axis_cross cross_test3a2
// Cross and 2 elements in the x axis
{
worksheet: cross;
no_scroll: true;
x: cross c1 c2;
y: a1 b2;
cross: b1 d1;
sparse: false;
}
```

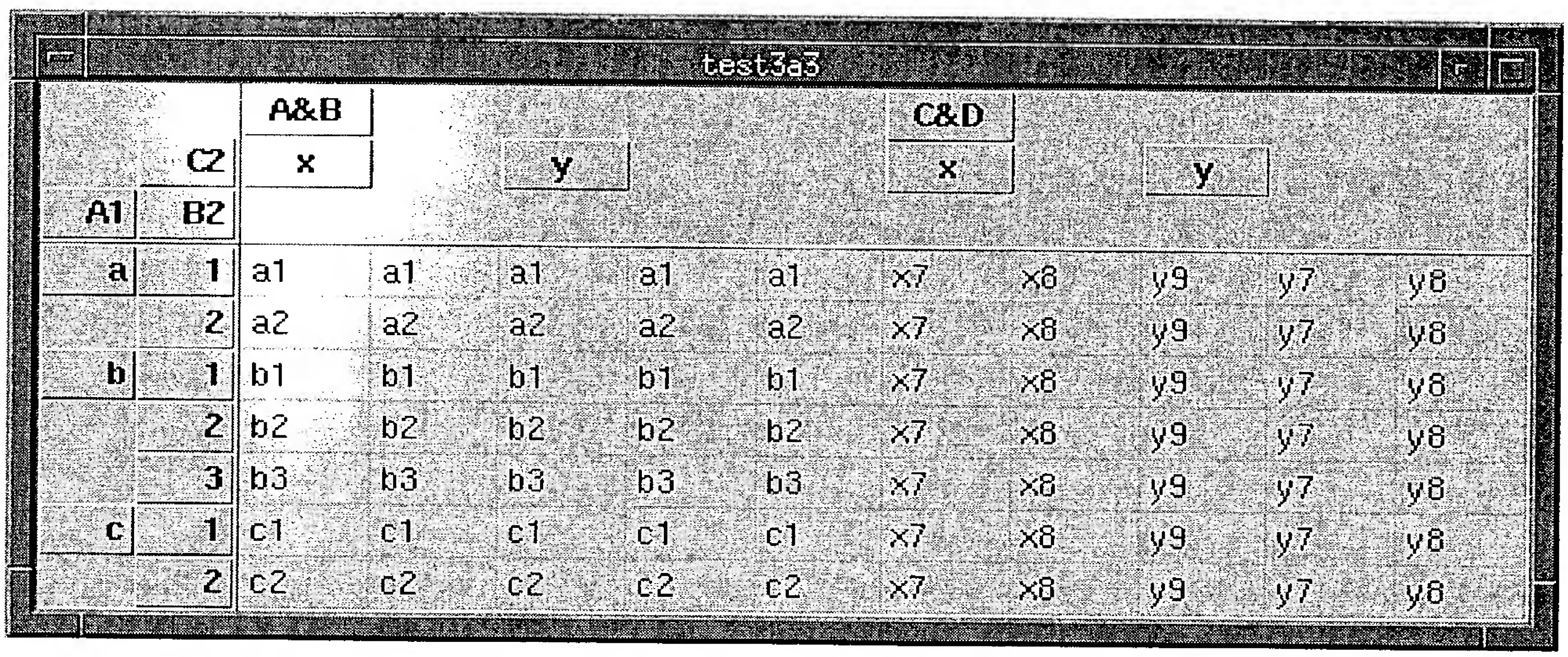

test3a2

		A&B	C&D	C&D	C&D	C&D	C&D
	C1		x	x	y	y	y
	C2		x	x	y	y	y
A1	B2						
a	1	a1	x7	x8	y9	y7	y8
a	2	a2	x7	x8	y9	y7	y8
b	1	b1	x7	x8	y9	y7	y8
b	2	b2	x7	x8	y9	y7	y8
b	3	b3	x7	x8	y9	y7	y8
c	1	c1	x7	x8	y9	y7	y8
c	2	c2	x7	x8	y9	y7	y8

F.2.10 Example Layout 10

This axis cross has items in the X, Y, and CROSS. The cross is displayed in the X-axis. There is no scrollbar present in this example.

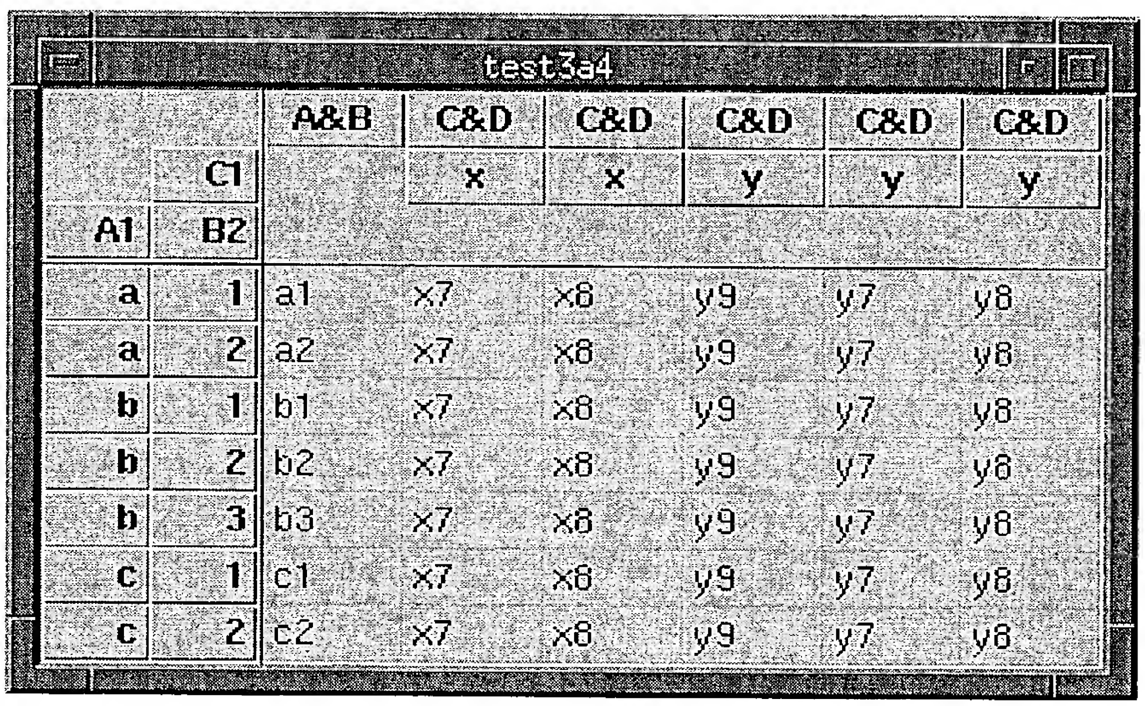
```
axis_cross cross_test3a3
{
no_scroll: true;
worksheet: cross;
x: cross c2;
y: a1 b2;
cross: b1 d1;
}
```



F.2.11 Example Layout 11

This example shows a layout with the CROSS and one element in the X-axis. Items are also specified in the Y, and the CROSS. No scrollbar is present and *sparse* is set to *false* (see the section on *Axis Cross Display* in *OIL Lesson 5*).

```
axis_cross cross_test3a4
{
no_scroll: true;
worksheet: cross;
x: cross c1;
y: a1 b2;
cross: b1 d1;
sparse: false;
}
```

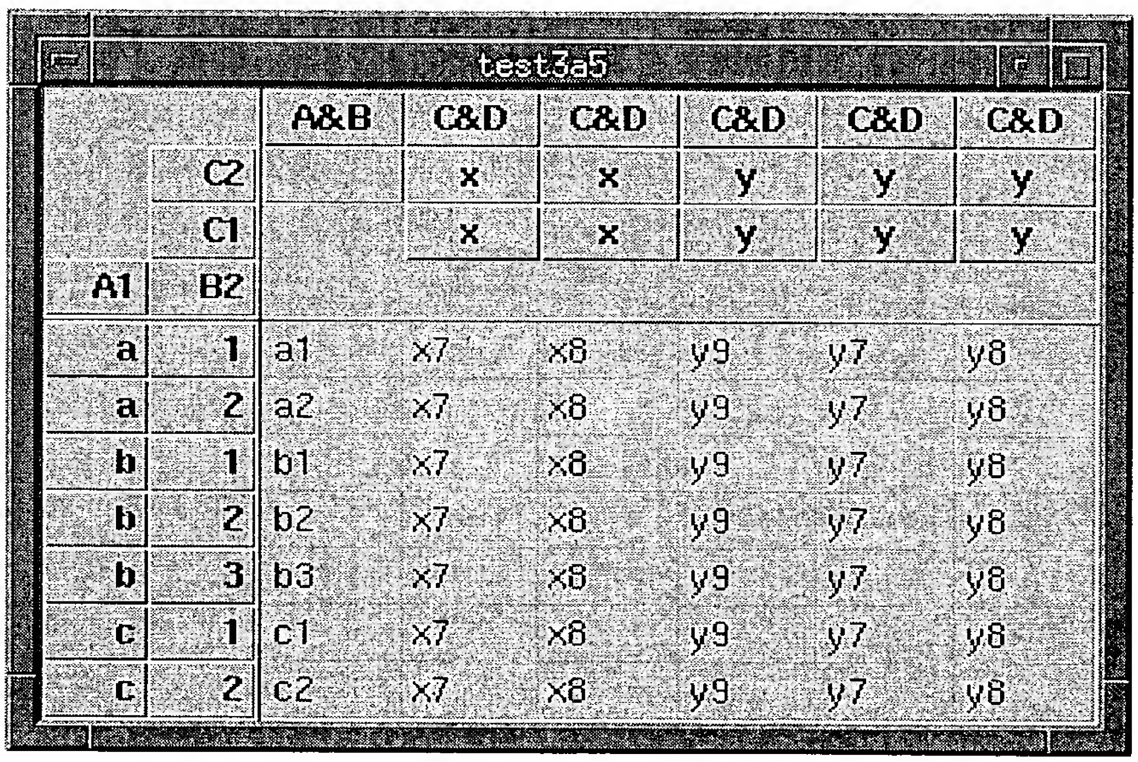


		A&B	C&D	C&D	C&D	C&D	C&D
	C1		x	x	y	y	y
A1	B2						
a	1	a1	x7	x8	y9	y7	y8
a	2	a2	x7	x8	y9	y7	y8
b	1	b1	x7	x8	y9	y7	y8
b	2	b2	x7	x8	y9	y7	y8
b	3	b3	x7	x8	y9	y7	y8
c	1	c1	x7	x8	y9	y7	y8
c	2	c2	x7	x8	y9	y7	y8

F.2.12 Example Layout 12

This example has the CROSS and two elements in the X-axis. The Y-axis and CROSS also contain items. There is no scrollbar and sparse is again set to false (see the section on *Axis Cross Display* in *OIL Lesson 5*).

```
axis_cross cross_test3a5
// Cross and 2 elements in the x axis
{
no_scroll: true;
worksheet: cross;
x: cross c2 c1;
y: a1 b2;
cross: b1 d1;
sparse: false;
}
```



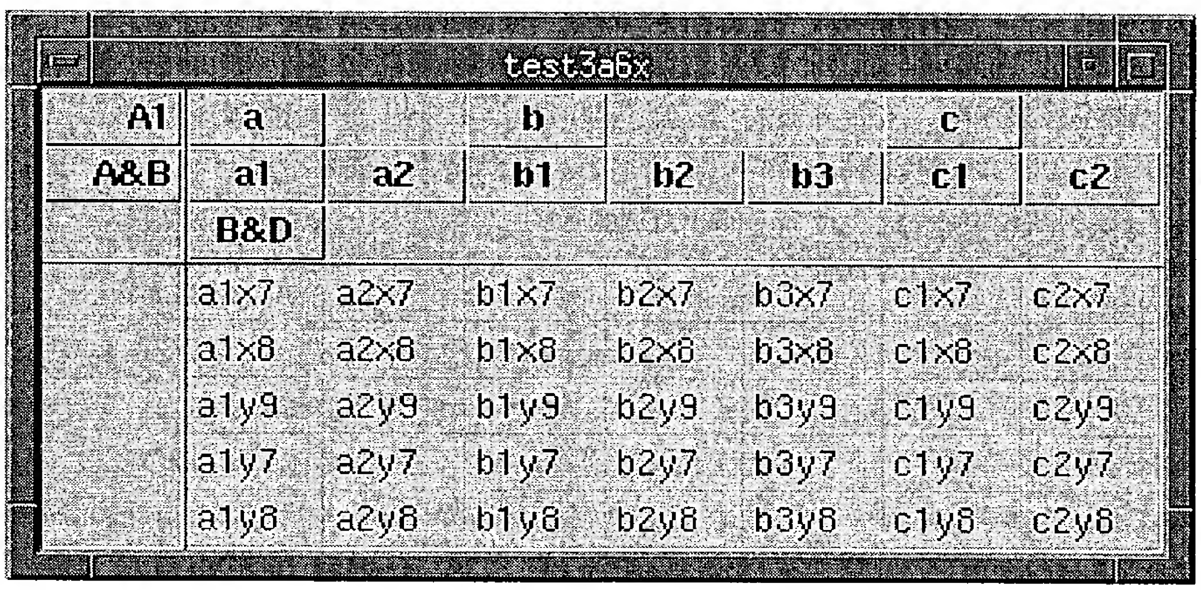
test3a5

		A&B	C&D	C&D	C&D	C&D	C&D
	C2		x	x	y	y	y
	C1		x	x	y	y	y
A1	B2						
a	1	a1	x7	x8	y9	y7	y8
a	2	a2	x7	x8	y9	y7	y8
b	1	b1	x7	x8	y9	y7	y8
b	2	b2	x7	x8	y9	y7	y8
b	3	b3	x7	x8	y9	y7	y8
c	1	c1	x7	x8	y9	y7	y8
c	2	c2	x7	x8	y9	y7	y8

F.2.13 Example Layout 13

This layout shows an axis cross with no items specified for the Y-axis (Y-axis empty). The cross is displayed in the X-axis and no scrollbar is present.

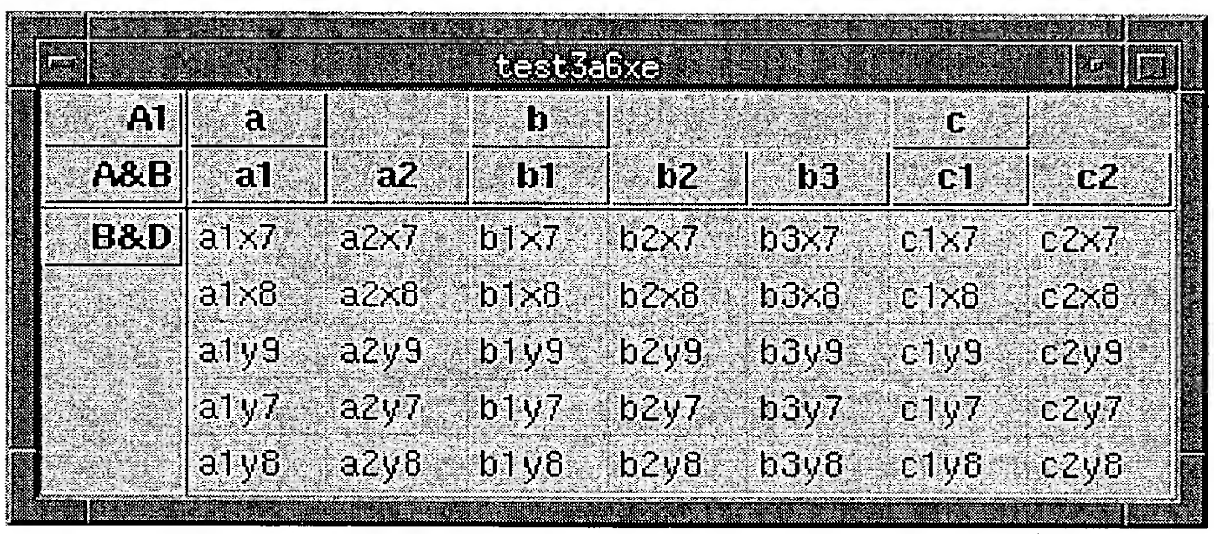
```
axis_cross cross_test3a6x
// empty y axis
{
no_scroll: true;
worksheet: cross;
x: a1 b1 cross;
cross: e1;
}
```



F.2.14 Example Layout 14

This example illustrates an axis cross with a combination of cross titles defaulting to the Y-axis and an empty Y-axis (no items in Y-axis).

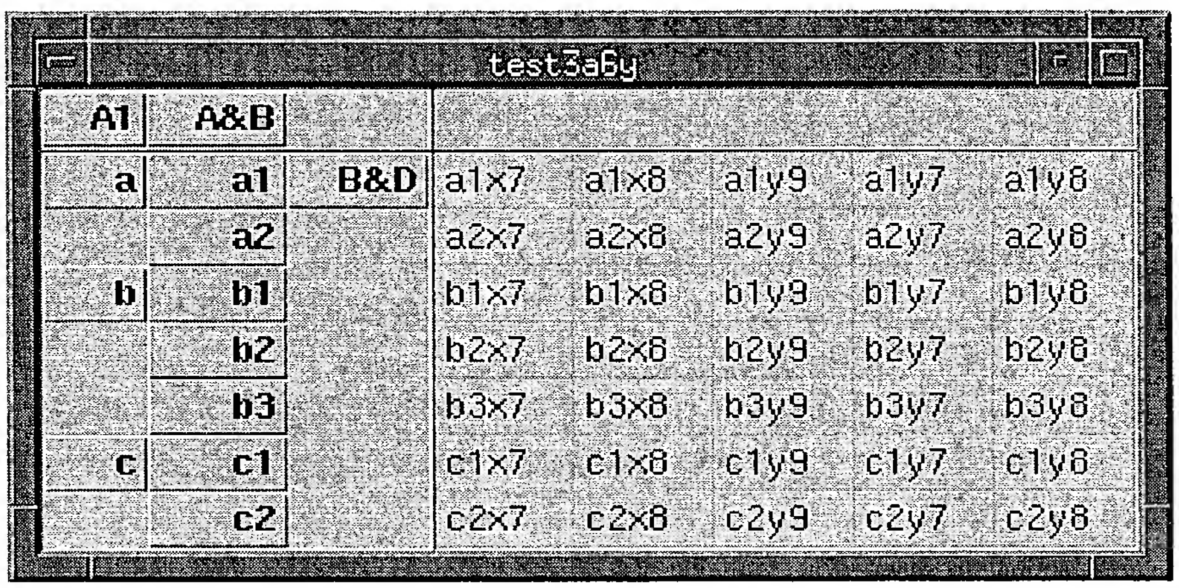
```
axis_cross cross_test3a6xe
// combo of cross titles default to y axis and
empty y axis
{
no_scroll: true;
worksheet: cross;
x: a1 b1;
cross: e1;
}
```



F.2.15 Example Layout 15

This axis cross layout displays an empty X-axis. There are items in the Y and the CROSS. No scrollbar is present.

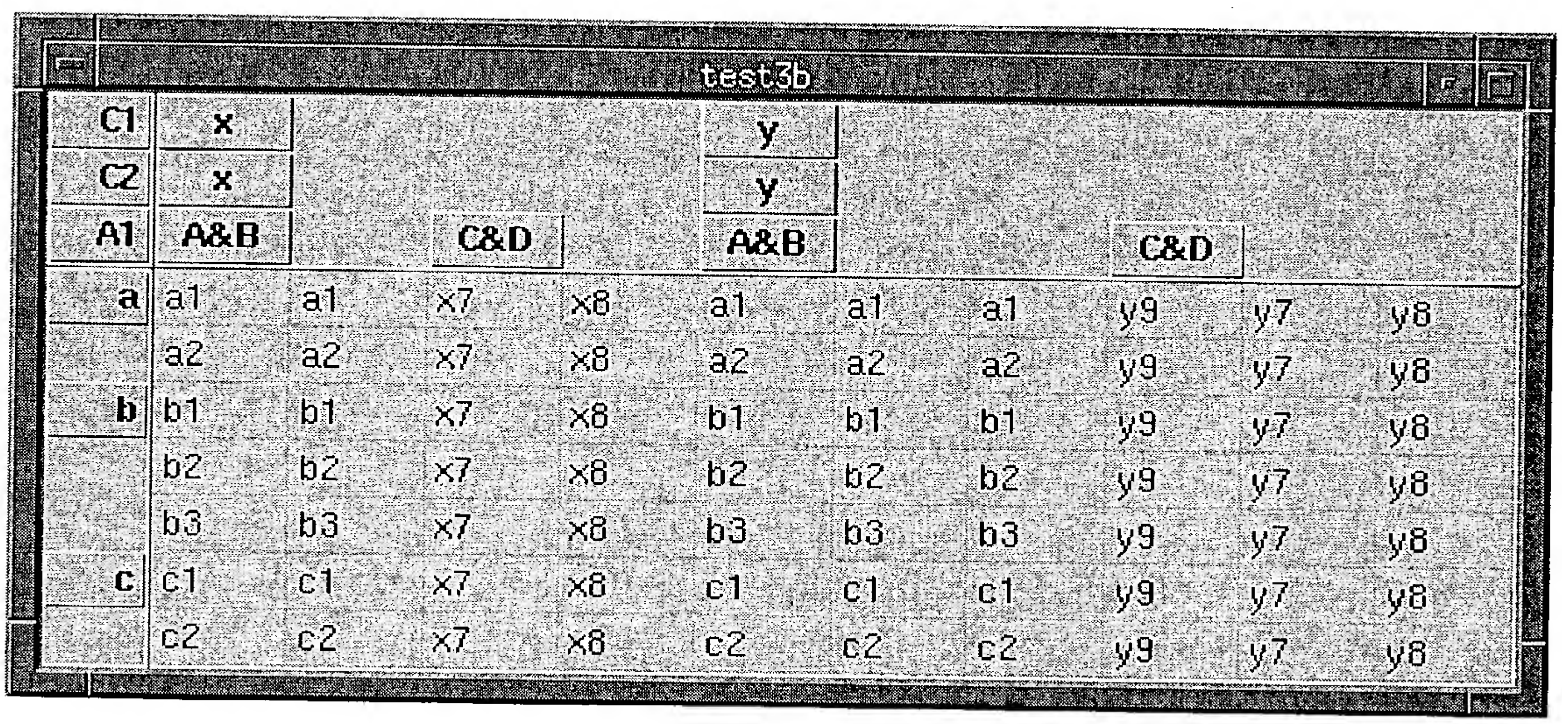
```
axis_cross cross_test3a6y
// empty x axis
{
no_scroll: true;
worksheet: cross;
y: a1 b1 cross;
cross: e1;
}
```



F.2.16 Example Layout 16

This example is more typical of axis cross having items in the X, Y, and CROSS. Again, no scrollbar is present and the cross is displayed in the X-axis.

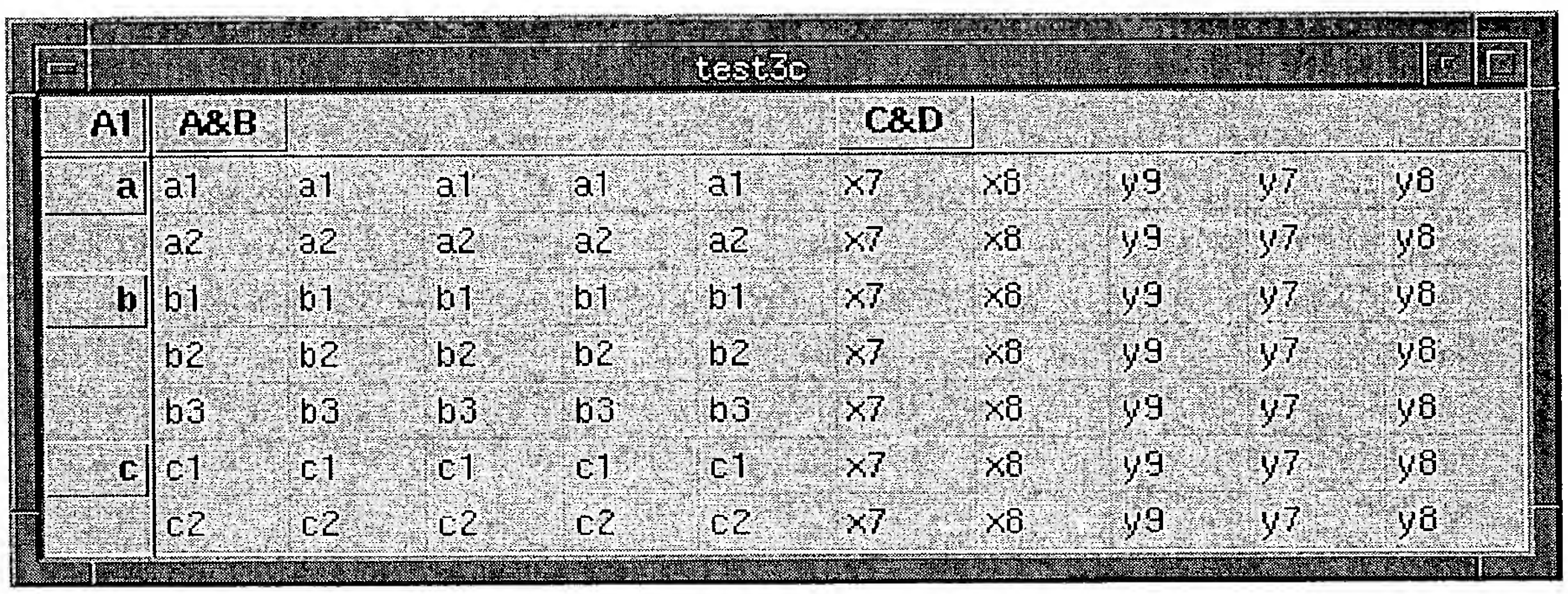
```
axis_cross cross_test3b
{
no_scroll: true;
worksheet: cross;
x: c1 c2 cross;
y: a1;
cross: b1 d1;
}
```



F.2.17 Example Layout 17

This example shows an axis cross with only the CROSS contained in the X-axis. No scrollbar is present.

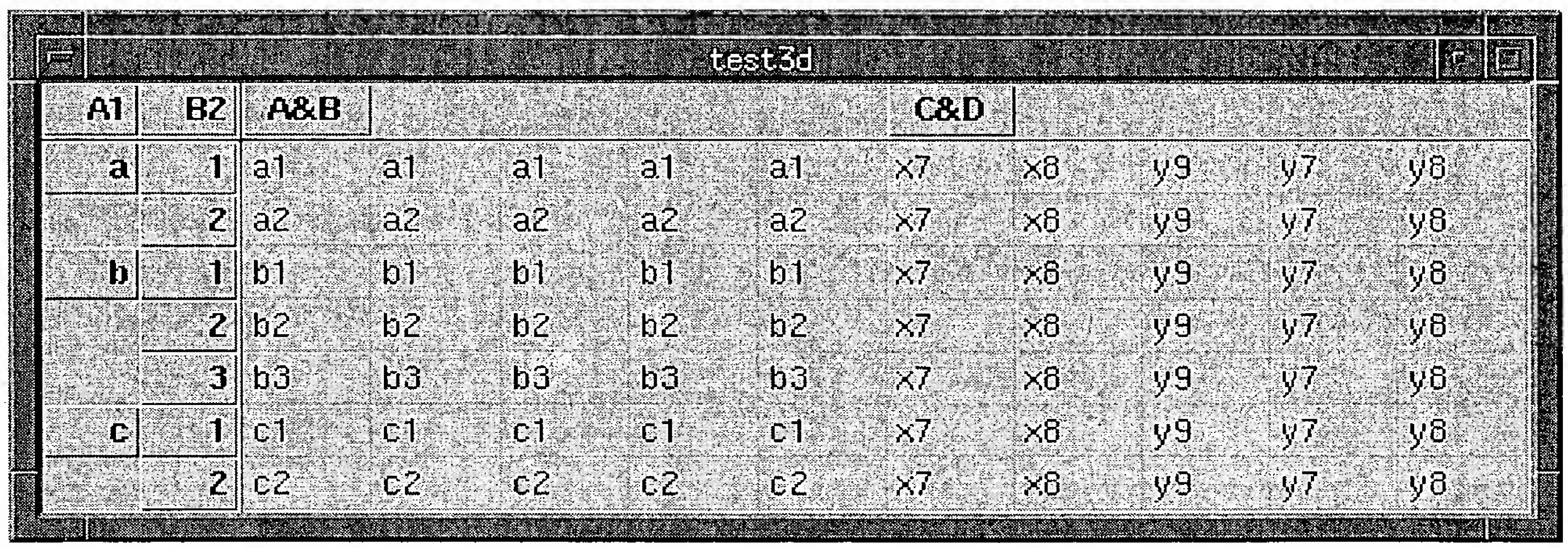
```
axis_cross cross_test3c
// Only the cross in the x axis
{
no_scroll: true;
worksheet: cross;
x: cross;
y: a1;
cross: b1 d1;
}
```



F.2.18 Example Layout 18

This axis cross example shows, again, a layout with only the CROSS in the X-axis.

```
axis_cross cross_test3d
// Only the cross in the x axis
{
no_scroll: true;
worksheet: cross;
x: cross;
y: a1 b2;
cross: b1 d1;
}
```



F.2.19 Example Layout 19

This example (shown on the next page) shows an axis cross with grouping (see *Axis Cross Layout Groups* in *OIL Lesson 5*). Only the Y-axis contains items.

```
axis_cross cross_test4
// Grouping test
{
no_scroll: true;
worksheet: cross;
y: a1 c1 b2 (b1 d1);
}
```

test4

A1	C1	B2		
a	x	1	A&B	a1
			C&D	x7
				x8
		2	A&B	a2
			C&D	x7
				x8
	y	1	A&B	a1
			C&D	y9
				y7
				y8
		2	A&B	a2
			C&D	y9
				y7
				y8
b	x	1	A&B	b1
			C&D	x7
				x8
		2	A&B	b2
			C&D	x7
				x8
		3	A&B	b3
			C&D	x7
				x8
	y	1	A&B	b1
			C&D	y9
				y7
				y8
		2	A&B	b2
			C&D	y9
				y7
				y8
		3	A&B	b3
			C&D	y9
				y7
				y8
c	x	1	A&B	c1
			C&D	x7

F.3 Example Set B

The next example illustrates an axis cross layout given the following worksheet:

```
replicating groups()
{
 variable alist = list("a", "b", "c");
 variable blist = list("1", "2", "1", "2", "3", "1", "2");
 variable clist = list("x", "y");
 variable dlist = list("7", "8", "9", "7", "8");

  [ A1 resource = alist.sublist(1,3); ];// Independent
  [ A1.title = "A1"; ];
  [ A2 = A1; ];
  [ A2.title = "A2"; ];
  [ A3 = upper(A1); ];
  [ A3.title = "upper(A1)"; ];

  [ B2 alts = if(A1=="A", blist.sublist(1,2),// F(a1)
                  if(A1=="b", blist.sublist(3,3),
                  blist.sublist(6,2))); ];
  [ B2.title = "B2"; ];
  [ B1 = A1 & B2; ];
  [ B1.title = "A&B"; ];
  [ C1 = clist.sublist(1,2); ];// Independent
  [ C1.title = "C1"; ];
  [ C2 = C1; ];
  [ C2.title = "C2"; ];
  [ D2 = if(C1=="x", dlist.sublist(1,2),// F(c1)
                  dlist.sublist(3,3)); ];
  [ D1 = C1 & D2; ];
  [ D1.title = "C&D"; ];
  [ E1 = B1 & D1; ];
  [ E1.title = "B&D"; ];
  [ E2 = if(E1 == "a1y8", list("yep", "FOB"),
                  nonexistent); ];
  [ E3 = if(E2 == "yep", "foo", nonexistent); ];
  [ F1 = upper(E1); ];
  [ F1.title = "upper(E)"; ];
  [ G1 = ""; ];
  [ G1.title = "independent group title"; ];
  [ h1 = upper(d1); ];
  [ h1.title = "upper(D1)"; ];
  [ i1 = upper(b1); ];
  [ i1.title = "upper(B1)"; ];
  [ J1 = ""; ];
  [ J1.title = "dependent group title"; ];
  [ K1 = nonexistent; ];[ K1.title = "Nonex1"; ];
```

```
  [ K2 = nonexistent; ];[ K2.title = "Nonex2"; ];
  [ K3 = nonexistent; ];[ K3.title = "Nonex3"; ];
  [ K4 = nonexistent; ];[ K4.title = "Nonex4"; ];
  [ K5 = nonexistent; ];[ K5.title = "Nonex5"; ];
  [ K6 = nonexistent; ];[ K6.title = "Nonex6"; ];
  [ M1 = do(D1,nonexistent); ]
  [ K7 = do(M1,nonexistent); ];[ K7.title = "Nonex7"; ];
}
```

F.3.1 Example Layout 1

This example illustrates using independent and dependent groups in axis cross (see *Axis Cross Layout Groups* in *OIL Lesson 5*).

```
axis_cross groups_test
{
no_scroll: true;
worksheet: groups;
y: a1 c1 b2 ("g1" b1 d1) ["j1" i1 h1];
// Independent_and_Dependent_Group_Examples - This
// layout is an example of independent and dependent
// groups in an axis specification.  The Y axis
// specification contains the independent group of b1
// and d1 with a title specified in g1.  It also
// contains the dependent group containing i1 and h1
// with a title coming from j1. See the comment at the
// top of this file, to run this example.
// See::Axis_Cross_Layout_Groups for an overview of
// this topic.
}
```

test

A1	C1	B2		independent group title	dependent group title
a	x	1	A&B	a1	A1
			C&D	x7	X7
				x8	X8
		2	A&B	a2	A2
			C&D	x7	X7
				x8	X8
	y	1	A&B	a1	A1
			C&D	y9	Y9
				y7	Y7
				y8	Y8
		2	A&B	a2	A2
			C&D	y9	Y9
				y7	Y7
				y8	Y8
b	x	1	A&B	b1	B1
			C&D	x7	X7
				x8	X8
		2	A&B	b2	B2
			C&D	x7	X7
				x8	X8
		3	A&B	b3	B3
			C&D	x7	X7
				x8	X8
	y	1	A&B	b1	B1
			C&D	y9	Y9
				y7	Y7
				y8	Y8
		2	A&B	b2	B2
			C&D	y9	Y9
				y7	Y7
				y8	Y8
		3	A&B	b3	B3
			C&D	y9	Y9
				y7	Y7
				y8	Y8
c	x	1	A&B	c1	C1

Index

P